1. 数据结构

数据结构或集合是按某种方式组织的数据集合，但是数据结构不仅仅是存储数据的容器，它们还提供了一系列的操作来访问（如搜索、插入、删除）和修改数据。
在CPT202中学习的时候，我们知道一个程序是由算法和数据结构组成的。要想开发出高性能的软件就需要优秀的算法和最佳的数据结构。
在面向对象编程（OOP）中，数据结构可以被视为包含其他对象的容器。这些容器对象（如 ArrayList、HashSet 等）可以包含任何类型的对象，这些对象被称为元素或成员。所以比如类也是一种数据结构，它定义了对象的结构和行为的蓝图，它封装了数据（属性或成员变量）和操作这些数据的方法。

1.1 Java集合框架（Java Collections Framework）

Java集合框架提供了多种数据结构，可以高效地组织和操作数据。
Java集合框架支持两种类型的容器：

1. Collection：用于存储一系列元素的容器。
   其还包含了大量子接口：
   List：存储有序的元素集合，允许重复元素。例如：ArrayList、LinkedList。
   Set：存储不重复的元素集合。例如：HashSet、TreeSet。
   Stack：存储对象的集合，后进先出（LIFO）的方式处理对象。
   Queue：存储对象的集合，先进先出（FIFO）的方式处理对象。
   PriorityQueue：存储对象的集合，按照优先级顺序处理对象。
2. Map：用于存储键值对的容器（在Python中称为字典）。

所有在Java集合框架中定义的接口和类都被组织在 java.util 包中。
Java集合框架的设计是使用接口、抽象类和具体类的一个优秀示例。
接口定义了集合框架的通用API。它们规定了类必须实现的方法，以确保不同集合类之间的互操作性。例如，Collection、List、Set、Map 等接口定义了相应的操作。
抽象类提供了接口的部分实现。这些类实现了接口中的一些方法，并可能添加一些额外的方法或字段，以支持更具体的功能。提供一个部分实现接口的抽象类使用户能够更方便地编写专门化容器的代码。
AbstractCollection 是一个便利抽象类，它提供了 Collection 接口的部分实现。例如，它提供了一些基本操作的实现，如 size()、isEmpty()、contains() 等。
具体类使用具体的数据结构来实现接口。这些类提供了接口方法的具体实现，以及可能的额外功能。例如，ArrayList、LinkedList、HashSet、HashMap 等都是具体类，它们实现了相应的接口，并使用了不同的数据结构来优化性能。

下图展示了Java集合框架（Java Collections Framework）中接口、抽象类和具体类之间层次结构。

Collection 接口是操作对象集合的根接口，即它是所有单列集合（如 List 和 Set）的父接口。
而AbstractCollection 是一个抽象类，它实现了 Collection 接口中的大部分方法，除了 add、size 和 iterator 方法。
Iterable 接口是一个标记接口，它有一个 iterator() 方法，用于返回一个 Iterator 对象。Iterator 接口提供了一种方法来顺序访问集合中的每个元素，而不需要暴露集合的底层表示。它定义了 hasNext()、next() 和 remove() 方法，用于遍历集合并可选地从集合中移除元素。因此我们可以使用for-each 循环通过 Iterator 或者数组索引来遍历元素。

下图展示了Iterable、Collection 和 Iterator 这三个接口的UML图。

下面代码演示了上面的一些方法。

```
import java.util.*;

public class TestCollection {
    public static void main(String[] args) {
        ArrayList<String> collection1 = new ArrayList<>();
        collection1.add("New York"); // add
        collection1.add("Atlanta");
        collection1.add("Dallas");
        collection1.add("Madison");

        System.out.println("A list of cities in collection1:");
        System.out.println(collection1);

        // the Collection interface's contains method
```

```
        System.out.println("\nIs Dallas in collection1? " +
collection1.contains("Dallas")); // contains

        // the Collection interface's remove method
        collection1.remove("Dallas"); // remove

        // the Collection interface's size method
        System.out.println("\n" + collection1.size() + // size
                " cities are in collection1 now");

        Collection<String> collection2 = new ArrayList<>();
        collection2.add("Seattle");
        collection2.add("Portland");

        System.out.println("\nA list of cities in collection2:");
        System.out.println(collection2);

        ArrayList<String> c1 = (ArrayList<String>) (collection1.clone()); //
clone
        c1.addAll(collection2); // addAll
        System.out.println("\nCities in collection1 or collection2:");
        System.out.println(c1);

        c1 = (ArrayList<String>) (collection1.clone());
        c1.retainAll(collection2); // retainAll
        System.out.println("\nCities in collection2:");
        System.out.println(c1);

        c1 = (ArrayList<String>) (collection1.clone());
        c1.removeAll(collection2); // removeAll
        System.out.println("\nCities in collection1, but not in 2: ");
        System.out.println(c1);
    }
}
```

结果如下，当然也可以在自己的ide中试验一下。

```
A list of cities in collection1:
[New York, Atlanta, Dallas, Madison]

Is Dallas in collection1? true

3 cities are in collection1 now

A list of cities in collection2:
[Seattle, Portland]

Cities in collection1 or collection2:
[New York, Atlanta, Madison, Seattle, Portland]

Cities in collection2:
[]

Cities in collection1, but not in 2:
[New York, Atlanta, Madison]
```

Java集合框架中的所有具体类（如 ArrayList、HashSet 等）通常实现了 java.lang.Cloneable 和 java.io.Serializable 接口。但是，java.util.PriorityQueue 类没有实现 Cloneable 接口。
在Java集合框架中，有些方法在 Collection 接口中定义，但在具体子类中可能无法实现。例如，对于只读集合（如由 Collections.unmodifiableCollection 方法返回的集合），你不能添加或删除元素。
当一个方法在当前上下文中不被支持时，可以通过抛出 UnsupportedOperationException 来明确告知调用者该操作不被支持。这是一种明确的错误处理方式，让调用者知道他们尝试执行了一个不被允许的操作。
示例如下。

```
public void someMethod() {
    throw new UnsupportedOperationException("Method not supported");
}
```

1.1.1 迭代器（Iterators）

在Java集合框架中，所有的集合类都实现了 Iterable 接口，这意味着它们都可以用迭代器进行遍历。
迭代器模式允许你顺序访问数据结构中的元素，而不需要暴露数据结构的内部存储细节。
我们可以使用增强型for循环（也称为for-each循环）遍历集合或数组。
示例如下。

```
for(String element : collection) {
    System.out.print(element + " ");
}
```

Collection 接口继承自 Iterable 接口，从而通过使用 Iterable 接口中的 iterator() 方法来获取一个集合的迭代器对象。这个 Iterator 对象提供了遍历集合的方法，如 hasNext()、next() 和 remove()。
下面的代码展示了如何使用Java集合框架中的 Iterator 来遍历 Collection 集合中的元素。

```
import java.util.*;

public class TestIterator {
    public static void main(String[] args) {
        Collection<String> collection = new ArrayList<>();
        collection.add("New York");
        collection.add("Atlanta");
        collection.add("Dallas");
        collection.add("Madison");

        Iterator<String> iterator = collection.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next().toUpperCase() + " ");
        }
        System.out.println();
    }
}
```

结果如下。

```
NEW YORK ATLANTA DALLAS MADISON
```

2. 列表（List）

下图展示了与 List 相关的接口、抽象类和具体类之间的层次结构。

List 接口表示的集合会按照元素插入的顺序来存储它们。这意味着元素在列表中有一个明确的顺序。
在 List 集合中，用户可以在添加元素时指定位置（索引），也可以在访问元素时通过索引来获取。
用户可以通过索引访问元素。
List 接口允许集合中的元素有相同的值，即允许重复元素。
Java集合框架中有两个具体类：ArrayList 和 LinkedList：

1. ArrayList：基于动态数组实现的列表，它提供了快速的随机访问。当需要频繁访问列表中的元素时，ArrayList 是一个好的选择。
2. LinkedList：基于链表实现的列表，它提供了快速的插入和删除操作。当需要在列表中频繁地添加或删除元素时，LinkedList 是一个更好的选择。
   它们都可以根据需要动态地增加或减少其容量。如果你的应用场景中只需要访问元素，而不需要频繁地插入或删除元素，那么使用数组可能是更高效的选择。数组提供了更快的随机访问能力，因为数组的索引访问是 O(1) 的时间复杂度，而 ArrayList 和 LinkedList 的随机访问时间复杂度分别是 O(1) 和 O(n)。

下图展示了 List 接口的UML图。

下图展示了 ListIterator 接口的UML图。

listIterator() 和 listIterator(startIndex)这两个方法返回 ListIterator 的实例，用于遍历列表。
ListIterator 接口扩展了 Iterator 接口，提供了双向遍历列表的能力，并且可以在遍历过程中向列表中添加元素。
nextIndex() 方法返回迭代器中下一个元素的索引。
previousIndex() 方法返回迭代器中上一个元素的索引。
add(E element) 方法将指定的元素插入到列表中，位置是在迭代器的当前位置之前，即下一个通过 next() 方法返回的元素之前。

2.1 ArrayList 类

下图展示了关于 ArrayList 类与 Collection 和 List 接口之间的继承关系。

ArrayList 是基于数组实现的，能够动态调整大小以适应添加和删除操作。
在插入新元素之前，需要将指定索引i之后的所有元素向右移动一位，以为新元素腾出空间。插入操作会使得 ArrayList 的大小增加1。
下图展示了这个过程。

2.2 LinkedList 类

下图展示了关于 LinkedList 类与 Collection 和 List 接口之间的继承关系。

LinkedList 由一系列节点组成，每个节点包含一个元素/值（element）和指向下一个节点的引用（next）。
从链表的头节点（head）开始，通过每个节点的 next 引用，可以遍历到链表的尾节点（tail），尾节点的 next 引用为 null。

下列代码定义了一个泛型节点类 Node，其中 E 表示节点可以存储任何类型的数据。

```
class Node<E> {
    E element;
    Node<E> next;

    public Node(E o) {
        element = o;
    }
}
```

2.3 对比

因此我们需要根据我们的应用场景和性能要求选择 ArrayList 或者 LinkedList 来存储和管理数据。如果我们需要经常通过索引访问，则可以选择 ArrayList ，如果是需要很多的插入和删除操作，则可以选择 LinkedList。

下面的代码展示了如何使用 ArrayList 和 LinkedList。

```
import java.util.*;

public class TestArrayAndLinkedList {
    public static void main(String[] args) {
        List<Integer> arrayList = new ArrayList<>();
        arrayList.add(1); // 1 is autoboxed to new Integer(1)
        arrayList.add(2);
        arrayList.add(3);
        arrayList.add(1);
        arrayList.add(4);

        arrayList.add(0, 10);
        arrayList.add(3, 30);

        System.out.println("A list of integers in the array list:");
        System.out.println(arrayList);

        LinkedList<Object> linkedList = new LinkedList<>(arrayList);
        linkedList.add(1, "red");
        linkedList.removeLast();
        linkedList.addFirst("green");

        System.out.println("Display the linked list backward with index:");
        for (int i = linkedList.size() - 1; i >= 0; i--) {
           System.out.print(linkedList.get(i) + " ");
       }
        System.out.println();

        System.out.println("Display the linked list forward:");
        ListIterator<Object> listIterator = linkedList.listIterator();
        while (listIterator.hasNext()) {
           System.out.print(listIterator.next() + " ");
       }
        System.out.println();

        System.out.println("Display the linked list backward:");
        listIterator = linkedList.listIterator(linkedList.size());
```

```
        while (listIterator.hasPrevious()) {
            System.out.print(listIterator.previous() + " ");
        }
    }
}
```

可以自己在ide上尝试下最后的结果。

```
A list of integers in the array list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list backward with index:
1 3 30 2 1 red 10 green
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backward:
1 3 30 2 1 red 10 green
Process finished with exit code 0
```

get(i) 方法可以在 LinkedList 中使用，但它是一个更耗时的操作，因为它需要从头开始遍历到指定索引 i 的位置才能找到元素。对于 ArrayList 来说，get(i) 方法是高效的，因为它是基于数组实现的，可以直接通过索引访问元素。
为了更高效地遍历列表，推荐使用迭代器或增强型for循环。这两种方法都不需要像 get(i) 那样从头遍历到指定索引，而是直接访问每个元素。
也就是使用下面的代码。

```
for (listElementType s : list) {
   process s;
}
```

代替原本基础的 for 循环和 get(i) 方法。

```
for (int i = 0; i < list.size(); i++) {
   process list.get(i);
}
```

2.4 静态方法

Collections 类提供了一组静态方法，用于对集合或列表执行各种操作。这些方法可以对集合或列表进行排序、搜索、修改等。
对于集合来说：

1. max 和 min 方法用于找出集合中的最大和最小元素。
2. disjoint 方法用于找出两个集合的不相交部分，即属于其中一个集合但不属于另一个集合的元素。
3. frequency 方法用于计算某个元素在集合中出现的次数。

对于列表来说：

1. sort 方法用于对列表进行排序。
2. binarySearch 方法用于在已排序的列表中进行二分查找。
3. reverse 方法用于反转列表中元素的顺序。
4. shuffle 方法用于打乱列表中的元素顺序。
5. copy 方法用于将一个列表的内容复制到另一个列表。
6. fill 方法用于用指定的元素填充列表。

对于 sort 方法，有两个重载版本。
static <T extends Comparable<? super T>> void sort(List<T> list)：这个版本的 sort 方法使用 Comparable 接口中的 compareTo 方法对列表进行排序。这意味着列表中的元素必须是 Comparable 的实例，或者其类必须实现了 Comparable 接口。
static <T> void sort(List<T> list, Comparator<T> c)：这个版本的 sort 方法使用 Comparator 接口中的 compare 方法对列表进行排序。这允许你定义自定义排序规则，即使元素不是 Comparable 的实例。

下图展示了 Collection 类的 UML 图。

还有一些其他的有用的静态方法如下：

1. rotate(List list, int distance)：
   这个方法将列表中的所有元素按照指定的距离进行旋转。如果距离为正数，则元素向右旋转；如果距离为负数，则元素向左旋转。例如，Collections.rotate(list, 2) 将列表中的元素向右移动两个位置。
2. replaceAll(List list, Object oldVal, Object newVal)：
   这个方法将列表中所有指定的旧值（oldVal）替换为新值（newVal）。这会影响列表中的所有匹配的元素。
3. indexOfSubList(List source, List target)：
   这个方法返回源列表（source）中第一个与目标列表（target）相等的子列表的索引。如果找不到匹配的子列表，则返回 -1。
4. lastIndexOfSubList(List source, List target)：
   这个方法返回源列表中最后一个与目标列表相等的子列表的索引。如果找不到匹配的子列表，则返回 -1。
5. swap(List list, int i, int j)：
   这个方法交换列表中指定位置（i 和 j）的元素。这两个位置的元素将被互换。
6. addAll(Collection<? super T> c, T... elements)：
   这个方法将指定数组中的所有元素添加到指定的集合（c）中。这允许你一次性向集合中添加多个元素。

2.4.1 sort() 方法

我们现在一个个查看这些关于列表的方法。
下面的代码展示了如何使用 Collections.sort() 对列表进行排序，这个排序是升序排序（自然排序），如果想要获得降序排序（逆自然顺序），可以使用 Collections.sort() 方法和 Collections.reverseOrder() 方法对列表进行降序排序。

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestCollectionsSort {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("red", "green", "blue");
        Collections.sort(list);
        System.out.println(list); // 输出: [blue, green, red]

        Collections.sort(list, Collections.reverseOrder());
        System.out.println(list); // 输出: [red, green, blue]
    }
}
```

2.4.2 binarySearch() 方法

这个方法会使用我们在算法种学习的二分查找去搜索对应的键。因此它要求我们的列表是升序排列的。如果要查找的键不在列表中，binarySearch 方法会返回一个负数，该数是键应该插入的位置（插入点 + 1）。
示例代码如下。

```
import java.util.Arrays;
import java.util.Collections;

public class TestBinarySearch {
    public static void main(String[] args) {
        List<Integer> list1 = Arrays.asList(2, 4, 7, 10, 11, 45, 50, 59, 60,
66);
        System.out.println("(1) Index: " + Collections.binarySearch(list1, 7));
 // 2
        System.out.println("(2) Index: " + Collections.binarySearch(list1, 9));
 // -4
        List<String> list2 = Arrays.asList("blue", "green", "red");
        System.out.println("(3) Index: " + Collections.binarySearch(list2,
"red")); // 2
        System.out.println("(4) Index: " + Collections.binarySearch(list2,
"cyan")); // -2
    }
}
```

2.4.3 reverse() 方法

这个方法可以反转列表中元素的顺序。
示例代码如下。

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestCollectionsReverse {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("yellow", "red", "green", "blue");
        Collections.reverse(list);
        System.out.println(list); // 输出: [blue, green, red, yellow]
    }
}
```

2.4.4 shuffle() 方法

这里的洗牌算法可以打乱列表中的元素顺序。
此外还可以使用 shuffle(List, Random) 方法，它允许使用一个特定的 Random 对象来随机重新排序列表中的元素。使用特定的 Random 对象来打乱另一个列表，如列表一样，Random 对象一样那么结果是一样的。
示例代码如下。

```
import java.util.Arrays;
```

```
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class TestCollectionsShuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("yellow", "red", "green", "blue");

        // 使用默认的随机数生成器来打乱列表
        Collections.shuffle(list);
        System.out.println(list);

        // 创建一个Random对象
        Random rnd = new Random();

        // 使用指定的Random对象来打乱列表
        List<String> anotherList = Arrays.asList("yellow", "red", "green",
"blue");
        Collections.shuffle(anotherList, rnd);
        System.out.println(anotherList);
    }
}
```

下面的代码展示了 Random 对象的种子相同，两个列表以相同的随机顺序被打乱。

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;

public class SameShuffle {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
        List<String> list2 = Arrays.asList("Y", "R", "G", "B");
        Collections.shuffle(list1, new Random(20));
        Collections.shuffle(list2, new Random(20));
        System.out.println(list1);
        System.out.println(list2);
    }
}
```

2.4.5 copy(dest, src) 方法

这个方法将源列表（src）中的所有元素的引用复制到目标列表（dest）中，保持相同的索引。
它实现的是浅拷贝操作，意味着它只复制元素的引用，而不是元素本身。如果源列表中的元素是可变对象，那么目标列表中的对应元素将引用同一个对象。
下面的代码展示了其的使用。

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestCollectionsCopy {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("yellow", "red", "green", "blue");
        List<String> list2 = Arrays.asList("white", "black");

        Collections.copy(list1, list2);
        System.out.println(list1); // 输出: [white, black, green, blue]
    }
}
```

如果我们将这两个列表的顺序修改一下，即目标列表（destination list）比源列表（source list）小，将会抛出运行时错误（IndexOutOfBoundsException）。
示例代码如下。

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestCollectionsCopy {
    public static void main(String[] args) {
        List<String> list2 = Arrays.asList("yellow", "red", "green", "blue");
        List<String> list1 = Arrays.asList("white", "black");

        Collections.copy(list1, list2);
    }
}
```

2.4.6 Arrays.asList() 方法

这个方法用于根据提供的变量长度参数列表创建一个新的 List。
这个方法返回一个 List 引用，指向 Arrays 类内部定义的一个类对象。这个内部类实际上是 ArrayList，但它是一个包装器（wrapper），用于将数组转换为列表。
以下是示例代码。

```
import java.util.Arrays;
import java.util.List;

public class TestArrayList {
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("red", "green", "blue");
        List<Integer> list2 = Arrays.asList(10, 20, 30, 40, 50);

        System.out.println(list1); // 输出: [red, green, blue]
        System.out.println(list2); // 输出: [10, 20, 30, 40, 50]
    }
}
```

2.4.7 nCopies(int n, Object o) 方法

这个方法会创建一个包含指定对象 o 的 n 个副本的列表。
使用 nCopies 方法创建的列表是不可变的，这意味着你不能向列表中添加或移除元素。
由于列表中的所有元素都是同一个对象的副本，它们都指向相同的内存地址。
返回的列表类型是 Collections$CopiesList，这是 java.util.Collections 类的一个内部类，它实现了 List 接口。
示例代码如下。

```
import java.util.Arrays;
import java.util.Calendar;
import java.util.Collections;
import java.util.List;

public class TestCollectionsNCopies {
    public static void main(String[] args) {
        List<Calendar> list1 = Collections.nCopies(3, new
GregorianCalendar(2022, Calendar.JANUARY, 1));

        System.out.println(list1);
    }
}
```

下面的代码展示了由于修改一个对象所以影响列表中的所有对象。

```
import java.util.*;

public class Test {
    public static void main(String[] args) {
        List<GregorianCalendar> list1 = Collections.nCopies(3, new
GregorianCalendar(2022, Calendar.JANUARY, 1));
        list1.get(0).set(Calendar.YEAR, 2024);
        for (GregorianCalendar g : list1) {
            System.out.println(g.get(Calendar.YEAR));
        }
    }
}
```

2.4.8 fill(List list, Object o) 方法

这个方法将列表 list 中的所有元素替换为指定的元素 o。
示例代码如下。

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestCollectionsFill {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("red", "green", "blue");
        Collections.fill(list, "black");
        System.out.println(list); // 输出: [black, black, black]
    }
}
```

2.4.9 max(), min() 方法

现在看关于 Collection 的方法。
Collections.max() 方法用于找出集合中的最大元素。它使用元素的 compareTo 方法来确定最大值。
Collections.min() 方法用于找出集合中的最小元素。它同样使用元素的 compareTo 方法来确定最小值。
示例代码如下。

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class TestCollectionsMaxMin {
    public static void main(String[] args) {
        List<String> collection = Arrays.asList("red", "green", "blue");
        System.out.println(Collections.max(collection)); // 输出: red
        System.out.println(Collections.min(collection)); // 输出: blue
    }
}
```

2.4.10 disjoint(collection1, collection2) 方法

这个方法用于检查两个集合是否有共同的元素。如果有共同元素，方法返回 false；如果没有共同元素，方法返回 true。
示例代码如下。

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class TestCollectionsDisjoint {
    public static void main(String[] args) {
        Collection<String> collection1 = Arrays.asList("red", "cyan");
        Collection<String> collection2 = Arrays.asList("red", "blue");
        Collection<String> collection3 = Arrays.asList("pink", "tan");

        System.out.println(Collections.disjoint(collection1, collection2)); // 输
出: false
        System.out.println(Collections.disjoint(collection1, collection3)); // 输
出: true
    }
```

```
}
```

2.4.11 frequency(collection, element) 方法

这个方法用于计算集合中某个元素出现的次数。
示例代码如下。

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class TestCollectionsFrequency {
    public static void main(String[] args) {
        Collection<String> collection = Arrays.asList("red", "cyan", "red");
        System.out.println(Collections.frequency(collection, "red")); // 输出: 2
    }
}
```

更多例子。

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class TestCollectionsFrequency {
    public static void main(String[] args) {
        Collection<String> collection = Arrays.asList(new String("red"), "cyan",
new String("red"), "red");
        System.out.println(Collections.frequency(collection, "red")); // 输出: 3
    }
}
```

"red" 被创建了三次（每次都是一个新的对象），但是由于 equals 方法用于比较字符串的内容，所以 "red" 被正确地计算为出现了三次。
同理如下。

```
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class TestCollectionsFrequency {
    public static void main(String[] args) {
        Collection<String> collection = Arrays.asList("red", "cyan", "red");
        System.out.println(Collections.frequency(collection, new
String("red"))); // 输出: 2
    }
}
```

3. Vector 和 Stack 类

这两个类在Java 2之前就已设计，在集合框架设计后，这些类被重新设计以适应Java集合框架，但为了兼容性，保留了它们的旧方法。

Vector 和 Stack 类在现代Java集合框架中不常用（因为它们的方法不是线程安全的，且功能已被 ArrayList、LinkedList 等类取代），但它们在集合框架中仍然占有一席之地，特别是为了保持向后兼容性。

3.1 Vector 类

Vector 类在功能上与 ArrayList 类似，但主要区别在于 Vector 类包含了同步（synchronized）方法，用于访问和修改向量（vector）。
同步方法可以防止数据损坏，当一个向量被两个或更多线程同时访问和修改时，同步机制确保了数据的一致性和完整性。
到目前为止讨论的所有类都不是同步的。
对于许多不需要同步的应用，使用 ArrayList 比使用 Vector 更高效。这是因为 ArrayList 不包含额外的同步开销。
addElement(Object element) 方法（从Java2保留的方法）与 add(Object element) 方法相同，唯一的区别在于 addElement 方法是同步的。
Vector 类的UML图如下。

3.2 Stack 类

Stack 类表示一个后进先出（LIFO, Last-In-First-Out）的栈对象集合。
在 Stack 中，所有元素的访问、检索和移除操作都只能从栈的顶部进行。
相关方法如下：

1. push(o: E) 方法用于在栈顶添加新元素。
2. peek() 方法用于获取栈顶元素，但不移除它。
3. pop() 方法用于移除栈顶元素，并返回该元素。
   Stack 类是 Vector 类的子类，这意味着它继承了 Vector 类的所有属性和方法，并在此基础上增加了栈特定的功能。
   empty() 方法（从Java2保留的方法）与 isEmpty() 方法相同，都用于检查栈是否为空。
   UML图如下。

4. 队列（Queues）和优先队列（Priority Queues）

队列是一种先进先出（FIFO, First-In-First-Out)的数据结构。
元素被添加到队列的末尾，并且从队列的开头被移除。
相关方法如下：

1. offer(o: E) 方法：这个方法用于向队列中添加一个元素。与 Collection 接口中的 add 方法相似，但对于队列来说，offer 方法是首选的方法。
2. poll 和 remove 方法：这两个方法用于从队列中移除元素。poll() 方法在队列为空时返回 null，而 remove() 方法在队列为空时抛出异常。
3. peek 和 element 方法：
   这两个方法用于查看队列中的元素而不移除它。peek() 方法在队列为空时返回 null，而 element() 方法在队列为空时抛出异常。
   在优先队列中，元素被分配优先级。当访问元素时，具有最高优先级的元素首先被移除。

4.1 队列（Queue）

Queue 接口继承自 Collection 接口，它包含了 Collection 接口的所有方法，并且添加了一些额外的操作。
其UML图如下。

LinkedList 类不仅可以作为列表使用，还可以作为队列使用，因为它实现了 Queue 和 Deque 接口。
下图展示了这些接口和类之间的继承关系。

Queue 接口支持在队列的两端插入和移除元素。而Deque 是 "double-ended queue" 的缩写，它扩展了 Queue 接口，增加了在队列两端插入和移除元素的方法。
Deque 接口定义了 addFirst(e)、removeFirst()、addLast(e)、removeLast()、getFirst() 和 getLast() 方法，即允许在队列的头部和尾部进行操作。
下面给出一个示例代码。

```
import java.util.LinkedList;
import java.util.Queue;

public class TestQueue {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.offer("Oklahoma");
        queue.offer("Indiana");
        queue.offer("Georgia");
        queue.offer("Texas");

        while (queue.size() > 0) {
            System.out.print(queue.remove() + " ");
        }
    }
}
```

它主要展示了 offer 方法的使用。

4.2 PriorityQueue 类

默认情况下，PriorityQueue 使用元素的自然排序（natural ordering）来确定优先级。这意味着元素必须实现 Comparable 接口，并且优先级是根据 compareTo 方法的结果来确定的。
值最小的元素被赋予最高的优先级，因此会首先从队列中移除。
如果有多个元素具有相同的最高优先级，这些元素之间的顺序是任意的。
可以通过 PriorityQueue 构造函数指定一个 Comparator 来定义元素的排序规则。
下图展示了其的UML图。

下面给出示例代码。

```
import java.util.*;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue<String> queue1 = new PriorityQueue<>();
        queue1.offer("Oklahoma");
        queue1.offer("Indiana");
```

```
        queue1.offer("Georgia");
        queue1.offer("Texas");
        System.out.println("Priority queue using Comparable:");
        while (queue1.size() > 0) {
            System.out.print(queue1.remove() + " ");
        } // 输出顺序：Georgia Indiana Oklahoma Texas

        PriorityQueue<String> queue2 = new PriorityQueue<>(4,
Collections.reverseOrder());
        queue2.offer("Oklahoma");
        queue2.offer("Indiana");
        queue2.offer("Georgia");
        queue2.offer("Texas");

        System.out.println("\nPriority queue using Comparator:");
        while (queue2.size() > 0) {
            System.out.print(queue2.remove() + " ");
        } // 输出顺序：Texas Oklahoma Indiana Georgia
    }
}
```

5. 练习

5.1 基础练习

用 Stack 类实现中缀表达式。（中缀表达式可以看算法的笔记）
示例代码如下。

```
import java.util.Stack;

public class EvaluateExpression {
    public static void main(String[] args) {
        // Check number of arguments passed
        if (args.length != 1) {
            System.out.println("Usage: java EvaluateExpression \"expression\"");
            System.exit(1);
        }
        try {
            System.out.println(evaluateExpression(args[0]));
        } catch (Exception ex) {
            System.out.println("Wrong expression: " + args[0]);
        }
    }

    /** Evaluate an expression */
    public static int evaluateExpression(String expression) {
        // Create operandStack to store operands
        Stack<Integer> operandStack = new Stack<>();
        // Create operatorStack to store operators
        Stack<Character> operatorStack = new Stack<>();
        // Insert blanks around (, ), +, -, /, and *
        expression = insertBlanks(expression);
        // Extract operands and operators
        String[] tokens = expression.split(" ");
```

```
        // Phase 1: Scan tokens
        for (String token : tokens) {
            if (token.length() == 0) // Blank space
                continue; // Back to the while loop to extract the next token
            else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
                // Process all +, -, *, / in the top of the operator stack
                while (!operatorStack.isEmpty() &&
                       (operatorStack.peek() == '+' ||
                       operatorStack.peek() == '-' ||
                       operatorStack.peek() == '*' ||
                       operatorStack.peek() == '/')) {
                    processAnOperator(operandStack, operatorStack);
                }
                // Push the + or - operator into the operator stack
                operatorStack.push(token.charAt(0));
            } else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
                // Process all *, / in the top of the operator stack
                while (!operatorStack.isEmpty() &&
                       (operatorStack.peek() == '*' ||
                        operatorStack.peek() == '/')) {
                    processAnOperator(operandStack, operatorStack);
                }
                // Push the * or / operator into the operator stack
                operatorStack.push(token.charAt(0));
            } else if (token.trim().charAt(0) == '(') {
                operatorStack.push('('); // Push '(' to stack
            } else if (token.trim().charAt(0) == ')') {
                // Process all the operators in the stack until seeing '('
                while (operatorStack.peek() != '(') {
                    processAnOperator(operandStack, operatorStack);
                }
                operatorStack.pop(); // Pop the '(' symbol from the stack
            } else { // An operand scanned
                // Push an operand to the stack
                operandStack.push(new Integer(token));
            }
        }
        // Phase 2: process all the remaining operators in the stack
        while (!operatorStack.isEmpty()) {
            processAnOperator(operandStack, operatorStack);
        }
        // Return the result
        return operandStack.pop();
    }

    /** Process one operator: Take an operator from operatorStack and
     * apply it on the operands in the operandStack */
    public static void processAnOperator(Stack<Integer> operandStack,
Stack<Character> operatorStack) {
        char op = operatorStack.pop();
        int op1 = operandStack.pop();
        int op2 = operandStack.pop();
        if (op == '+')
            operandStack.push(op2 + op1);
        else if (op == '-')
```

```
            operandStack.push(op2 - op1);
        else if (op == '*')
            operandStack.push(op2 * op1);
        else if (op == '/')
            operandStack.push(op2 / op1);
    }

    public static String insertBlanks(String s) {
        String result = "";
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
                    s.charAt(i) == '+' || s.charAt(i) == '-' ||
                    s.charAt(i) == '*' || s.charAt(i) == '/') {
                result += " " + s.charAt(i) + " ";
            } else
                result += s.charAt(i);
        }
        return result;
    }
}
```

5.2 进阶练习

5.2.1 比较迭代器（Iterator）和使用 get() 方法

1. 创建一个包含100,000个 Integer 对象的 LinkedList。
2. 使用 System.currentTimeMillis() 创建一个计时器.
3. 使用 get() 方法遍历 LinkedList。
4. 使用迭代器遍历 LinkedList。
5. 输出上述两个时间值。

示例代码如下。

```
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;

public class ListIteratorTest {
    public static void main(String[] args) {
        // 创建一个包含100,000个Integer对象的LinkedList
        List<Integer> list = new LinkedList<>();
        for (int i = 0; i < 100000; i++) {
            list.add(i);
        }

        // 创建一个计时器
        long startTimeGet = System.currentTimeMillis();

        // 使用get()方法遍历LinkedList
        for (int i = 0; i < list.size(); i++) {
            list.get(i);
        }
        long endTimeGet = System.currentTimeMillis();
```

```
        System.out.println("Time consumed by get(): " + (endTimeGet -
startTimeGet) + " ms");

        // 创建一个计时器
        long startTimeIterator = System.currentTimeMillis();

        // 使用迭代器遍历LinkedList
        Iterator<Integer> iterator = list.iterator();
        while (iterator.hasNext()) {
            iterator.next();
        }
        long endTimeIterator = System.currentTimeMillis();
        System.out.println("Time consumed by iterator(): " + (endTimeIterator -
startTimeGet) + " ms");
    }
}
```

5.2.2 优先队列

创建两个优先队列（Priority Queues），并展示它们的并集（union）、差集（difference）和交集（intersection）。

1. 首先，创建两个 PriorityQueue 实例，并向它们添加指定的元素。
2. 使用 PriorityQueue 的 addAll() 方法将第二个优先队列中的所有元素添加到第一个优先队列中，从而得到两个队列的并集。
3. 使用 PriorityQueue 的 removeAll() 方法从第一个优先队列中移除第二个优先队列中存在的所有元素，从而得到两个队列的差集。
4. 使用 PriorityQueue 的 retainAll() 方法从第一个优先队列中移除第二个优先队列中不存在的所有元素，从而得到两个队列的交集。

示例代码如下。

```
import java.util.PriorityQueue;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        // 创建两个优先队列并添加元素
        PriorityQueue<String> queue1 = new PriorityQueue<>();
        PriorityQueue<String> queue2 = new PriorityQueue<>();

        String[] elements1 = {"George", "Jim", "John", "Blake", "Kevin",
"Michael"};
        String[] elements2 = {"George", "Katie", "Kevin", "Michelle", "Ryan"};

        for (String element : elements1) {
            queue1.add(element);
        }
        for (String element : elements2) {
            queue2.add(element);
        }

        // 获取并集
        PriorityQueue<String> union = new PriorityQueue<>(queue1);
        union.addAll(queue2);
        System.out.println("Union: " + union);
```

```
        // 获取差集
        queue1.removeAll(queue2);
        System.out.println("Difference (queue1): " + queue1);

        // 获取交集
        queue2.retainAll(queue1);
        System.out.println("Intersection (queue2): " + queue2);
    }
}
```

5.2.3 Stack 类

使用 Stack 识别并验证圆括号 ()、花括号 {} 和方括号 [] 的正确配对，确保它们不会重叠。例如 (a{b)} 是非法的。（乐扣题目）
示例代码如下。

```
import java.util.Stack;

public class BracketChecker {
    public static void main(String[] args) {
        // 从键盘读取一行程序源代码
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        System.out.println("Enter a program source-code line:");
        String input = scanner.nextLine();

        // 检查括号是否配对
        System.out.println(isPaired(input) ? "Paired" : "Unpaired");
    }

    public static boolean isPaired(String expression) {
        Stack<Character> stack = new Stack<>();
        for (int i = 0; i < expression.length(); i++) {
            char ch = expression.charAt(i);
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            } else if (ch == ')' || ch == '}' || ch == ']') {
                if (stack.isEmpty() || stack.pop() != getMatching(ch)) {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }

    public static char getMatching(char ch) {
        switch (ch) {
            case '(': return ')';
            case '{': return '}';
            case '[': return ']';
            case ')': return '(';
            case '}': return '{';
            case ']': return '[';
            default: return 0;
```

```
        }
    }
}
```