JAVASE-DAY03/DAY04 晚课

1. equlas 方法在 contains 和 remove 中的作用

- 在包含（contains）和移除（remove）方法中，`equals` 方法的作用是用于确定一个对象是否等于另一个对象。
- 在使用 `contains` 方法时，它会内部调用每个元素的 `equals` 方法来比较目标对象与集合中的每个元素是否相等。`equals` 方法的实现决定了对象之间相等的条件。如果没有对 `equals` 方法进行自定义的重写，默认情况下它会使用对象的引用相等性（即两个对象引用同一内存地址）进行比较。
- 在使用 `remove` 方法时，它也会依赖于 `equals` 方法来确定要移除的对象。它会在集合中查找与给定对象相等的元素，并将其移除。如果没有重写 `equals` 方法，默认情况下它会使用对象的引用相等性来进行比较，这可能会导致无法正确删除对象。

```
Java
package cn.tedu.collection;

import java.util.ArrayList;
import java.util.Objects;


public class Demo {
    public static void main(String[] args) {
        ArrayList<Student> students = new ArrayList<>();
        //往集合中存储 3 个对象
        students.add(new Student("张三",18));
        students.add(new Student("李四",28));
        students.add(new Student("王五",38));

        //声明一个学生对象判断是否包含在 students 集合中
        Student stu = new Student("张三", 18);
```

```
        /**
         *  boolean contains 方法 是否包含
         *  如果包含，则返回 true
         *  如果不包含，则返回 false
         *  contains 的判断标准是通过给定的元素和集合中
         *  现有的元素逐一进行 equals 比较，
         *  只要返回为 true，则表示包含
         *  只要返回为 false，则表示不包含
         */
        //重写 equals 之前返回 false
        boolean contains = students.contains(stu);
        System.out.println("是否包含："+contains);

        /**
         * boolean remove(E e) 移除元素
         * 从集合中删除与指定元素进行 equals 比较为 true 的元素
         * 注意：只会删除一个，是从集合中顺序比较，删除第一个 equals
         * 比较为 true 的元素，删除后立即停止后面的操作
         */

        //往集合中添加张三元素
        students.add(stu);
        System.out.println(students);
        System.out.println("-------删除后");
        students.remove(stu);
        System.out.println(students);

    }
}
class Student{
    String name;
    int age;
    public Student(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;
        Student student = (Student) o;
        return age == student.age && Objects.equals(name,
student.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, age);
    }

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

因此，在使用包含和移除方法时，重写`equals`方法非常重要，以便根据对象的属性或内容来定义对象之间的相等性，确保正确的行为。

2. 集合保存对象的引用

集合中存放的是对象的引用,并不是对象的复制品

```
Java
package cn.tedu.collection;

import java.util.ArrayList;
import java.util.Collection;

public class StudentDemo {
    public static void main(String[] args) {
        Collection<Student> students = new ArrayList<>();
        Student stu = new Student("迪丽热巴", 28);
        students.add(stu);
```

```
        System.out.println("学生对象："+stu);
        System.out.println("集合："+students);
        /**
         * 重新修改对象的属性
         * 总结：
         * 集合中存放的是对象的引用
         * 如果对对象的属性进行修改，同样会影响集合中对象的属性
         * 因为指向的是同一个对象
         */
        stu.setName("杨幂");
        System.out.println("学生对象："+stu);
        System.out.println("集合："+students);
    }
}
```

3. 集合和数组的转换

集合转为数组

List 的 toArray 方法用于将集合转换为数组。但实际上该方法是在 Collection 中定义的，所以所有的集合都具备这个功能。其有两个方法:

```
Java
Object[] toArray()
<T>T[] toArray(T[] a)
```

其中第二个方法是比较常用的，我们可以传入一个指定类型的数组，该数组的元素类型应与集合的元素类型一致。返回值则是转换后的数组，该数组会保存集合中所有的元素。

```
Java
package cn.tedu.collection;

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

```
public class ArrayDemo {
    public static void main(String[] args) {
        Collection<Integer> nums =new ArrayList<Integer>();
        for (int i = 0; i <10 ; i++) {
            nums.add(i);
        }
        System.out.println("集合元素："+nums);
        /**
         将集合转换为数组
         toArray(); 不常用
         */
        Object[] objects = nums.toArray();
        System.out.println("数组元素："+Arrays.toString(objects));
        /**
         *  <T> T[] toArray(T[] a);
         *  可以传入一个指定类型的数组
         *  但是该数组的元素类型应该与集合中元素类型一致  否则会抛出数组存储类型不兼容的异常
         *  返回值则是转换后的数组,该数组会保存集合中的所有元素
         *
         *  原理：
         *  在执行 toArray(new Integer[40])方法时,
         *  1. 会默认在底层创建该长度的数组
         *  2. 遍历集合元素，依次存储到数组中
         *  如果数组长度超过集合的长度,剩余元素则自动用默认值填充
         *  如果数组长度小于集合的长度,则又创建一个新数组，类型和数组相同，大小和集合相同
         *
         */
//      String[] strings = nums.toArray(new String[10]); 报错
        Integer[] i1=nums.toArray(new Integer[40]);  //数组长度>集合长度 默认值填充
        System.out.println(Arrays.toString(i1));
```

```
        Integer[] i2 = nums.toArray(new Integer[5]); //不会进行截取
        System.out.println(Arrays.toString(i2));
    }
}
```

数组转为集合

Arrays 类中提供了一个静态方法 asList，使用该方法我们可以将一个数组转换为对应的 List 集合。

返回的 List 的集合元素类型由传入的数组的元素类型决定。

```
Java
static <T>List<T> asList<T… a>
```

需要注意的是，返回的集合我们不能对其增删元素，否则会抛出异常。并且对集合的元素进行的修改会影响数组对应的元素。

```
Java
package cn.tedu.collection;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        String[] arr = {"张三","李四","王五","赵六"};
        System.out.println("数组元素："+Arrays.toString(arr));

        /**
         * 将数组转为集合
         * <T> List<T> asList(T... a)
         */
        List<String> strings = Arrays.asList(arr);
        System.out.println("集合元素："+strings);
        /**
         * 对集合的操作就是对源数组的操作
         */
```

```
        strings.set(1, "2");
        System.out.println("修改后的 list 集合:"+strings);
        System.out.println("修改后的 arr 数组
:"+Arrays.toString(arr));
        /**
         * 从数组转换过来的集合是不能添加新元素的
         * 否则会抛出不受支持的操作异常
         * UnsupportedOperationException
         * 因为对集合元素操作就是对源数组操作
         * 添加元素会导致数组扩容,从而表示不了源数组
         */
//         strings.add("翠花");


        //如果想添加新元素,需要自行创建一个集合
        List<String> list2 = new ArrayList<String>(); //也可以直接
在()中加入 list
        list2.addAll(strings);
        System.out.println("list2:"+list2);
        list2.add("4");
        System.out.println("修改后的 list2:"+list2);
    }
}
```

4. Lambda 表达式

4.1 语法特点

Lambda 表达式，也可称为闭包，它是推动 Java 8 发布的最重要新特性。在 Java 中，Lambda 表达式是一种函数式编程的语法特性，用于简化函数式接口的实现。Lambda 表达式可以用来创建匿名函数或简洁地表示函数式接口的实现。

```
Java
(参数列表) -> {方法体}
```

- "参数列表" 可以是多个或者 0 个，参数的类型可以显示声明，也可以根据上下文推断，如果没有参数，直接写( )
- -> 是 Lambda 表达式的箭头操作符，用于分隔参数列表和方法体。
- {方法体} 是 Lambda 表达式的方法体，可以是一个表达式或一段代码块。如果方法体只有一条表达式，可以省略花括号 `{}`。如果方法体是一段代码块，则需要使用花括号包裹，并且需要使用 `return` 关键字来返回结果。

4.2 何时使用

Lambda 表达式主要用于简化函数式编程，当满足以下条件时可以使用 Lambda 表达式：

- 代码上下文要求使用函数式接口（Functional Interface），即只包含一个抽象方法的接口。
- Lambda 表达式的参数类型可以显式声明，也可以根据上下文进行推断。
- Lambda 表达式的方法体可以是一个表达式或一段代码块。

4.3 {} 的省略原则

在 Lambda 表达式中，如果方法体只有一条表达式，可以省略花括号 `{}`。这种情况下，表达式的结果将作为 Lambda 表达式的返回值。

```
Java
Function<Integer, Integer> square = x -> x * x;  // 省略了花括号
```

4.4 参数列表中 "()" 的省略原则：

在 Lambda 表达式中，如果参数列表为空或只有一个参数，可以省略参数列表中的圆括号 `()`。但当参数列表中有多个参数时，不能省略圆括号。

```
Java
// 参数列表为空
Runnable runnable = () -> System.out.println("Hello Lambda!");
```

```
// 参数列表只有一个参数
Consumer<String> printer = s -> System.out.println(s);

// 参数列表有多个参数
BinaryOperator<Integer> add = (a, b) -> a + b;
```

5. ArrayList 和 LinkedList 的区别（需要背）

ArrayList 和 LinkedList 是 Java 集合框架中的两个常用列表（List）实现，它们之间有以下区别：

1. 底层数据结构：ArrayList 底层使用数组实现，而 LinkedList 底层使用双向链表实现。
2. 访问效率：ArrayList 在随机访问元素时效率较高，因为它可以根据索引直接定位到元素的位置，而 LinkedList 在随机访问时需要从头或尾开始遍历链表
3. 插入和删除效率：LinkedList 在插入和删除元素时效率较高，因为它只需要调整链表中的指针，而 ArrayList 在插入和删除元素时需要进行数据的移动和复制。
4. 内存占用：LinkedList 在每个元素中存储了额外的链表指针，因此在存储大量元素时会占用较多的内存。ArrayList 则直接使用数组存储元素，不需要额外的指针，因此内存占用相对较少。
5. 迭代性能：LinkedList 在迭代操作（如使用迭代器或 for-each 循环）时性能较差，因为每次迭代都需要从头或尾开始遍历链表。ArrayList 则可以通过索引快速定位元素，迭代性能较好。

综上所述，ArrayList 适用于随机访问和读取操作较多的场景，而 LinkedList 适用于频繁的插入、删除的场景。根据具体的需求，可以选择适合的列表实现。