Java基础

zcx

一、面向对象高级

1. **代码块**

- 是类中的五大成分之一(成员变量、构造器、方法、代码块、内部类)
- 代码块分为两种：静态代码块、实例代码块
  - **静态代码块**：**格式：static{}**。特点：类加载时执行，只执行一次。作用：完成类的初始化
  - ```
    public class test {
    public static String a; 静态成员变量
    public static int[] arr=new int[52];
    static{
    // 静态代码块，与类一起优先加载，只执行一次，先加载静态代码块
    //故常用于初始化类的静态成员变量
    System.out.println("静态代码块");
    a = "ssssss";
    arr[0]=1; //初始化静态数组
    }
    public static void main(String[] args) {
    System.out.println("main方法");
    }
    }
    ```
  - **实例代码块**：**格式：{}**。特点：对象创建时执行，每创建一个对象执行一次。作用：完成对象的初始化
  - ```
    public class test2 {
    private String name;
    { // 实例代码块，无static修饰，属于对象
    System.out.println("实例代码块");
    name = "cxz"; //初始化对象的实例资源
    }
    public static void main(String[] args) {
    test2 t = new test2();
    System.out.println("main方法");
    ```

```
}
}
```

2.**内部类**

- 如果一个类定义在另一个类的内部，则称这个类位内部类
- **成员内部类**：就是类中的一个普通成员。外部类对象持有
  - 成员内部类可以直接访问外部类的静态成员，也可以直接访问外部类的实例成员
  - 成员内部类的实例方法中，可以直接拿到当前寄生的外部类对象 语法：外部类名.this
  - ```
    public class classd { //外部类
    public static int a=0; 外部类的静态成员
    private int b=0; 外部类的实例成员
    public class innerclass { //成员内部类，属于外部类对象
    public void show()
    {
    System.out.println("show");
    System.out.println(a);
    System.out.println(b);
    System.out.println(classd.this);
    }
    }
    }
    public class test {
    public static void main(String[] args)
    {
    //创建对象：外部类名.内部类名 创建内部类对象名 = new 外部类名().new 内部类名();
    classd.innerclass a = new classd().new innerclass();
    a.show();
    }
    }
    ```

- **静态内部类**：有static修饰的内部类，属于外部类自己持有
  - 静态内部类能直接访问外部类的静态成员
  - 静态内部类中不可以访问外部类的实例成员
  - ```
    public class classd { 外部类
    public static String name="aaaa"; 外部类静态成员变量
    public static class innerclass{ 静态内部类
    public void show(){
    System.out.println(name);
    }
    }
    }
    ```

```
public class test {
public static void main(String[] args)
{//创建对象：外部类名.内部类名 对象名 = new 外部类名.内部类名()
classd.innerclass ci = new classd.innerclass();
ci.show();
}
}
```

- **局部内部类**：定义在方法中、代码块中、构造器等执行体中(基本没用)
- **匿名内部类**：是一种特殊的局部内部类
  - 匿名是指：程序员不需要为这个类声明名字，**默认有个隐藏的名字(外部类名$编号.class)**
  - **格式：new 抽象父类/抽象接口(){类体，一般是方法重写}；**
  - 特点：**匿名内部类本质是一个子类，并会立即创建出一个子类对象**
  - 作用：更方便的创建一个子类对象
  - ```
    public abstract class animal { 抽象父类
    public abstract void cry();
    }
    public class test { 主类
    public static void main(String[] args) {
    animal a = new animal(){ 匿名内部类
    @Override
    public void cry() {
    System.out.println("cry");
    }
    };别忘了要写上分号
    }这部分就是匿名内部类，相当于一个子类，同时还是一个子类对象，就跟创建一个子类继承父类然后声明对象是一样的
    }编译看左边，运行看右边，是可以运行的
    ```
  - **常见使用形式：通常作为一个对象参数传输给方法**
  - ```
    public class test2 { 主类
    public static void main(String[] args) {
    swim s1 = new swim(){ 第一个匿名内部类
    @Override
    public void swiming() {
    System.out.println("student can swiming");
    }
    };
    start(s1); 调用方法
    start(new swim(){ 第二个匿名内部类,现在是直接充当了start需要的参数，因为内部类能立即创建出一个子类对象
    @Override
    ```

```
public void swiming() {
System.out.println("teacher can swiming");
}
});
} 上述这俩就是匿名内部类不同的使用形式
public static void start(swim s){ start方法
s.swiming();
}
}
interface swim{
void swiming(); 抽象方法
}
```

- **应用场景**
  - 调用别人提供的方法实现需求时，这个方法正好可以让我们传输一个匿名内部类对象给其使用(不是主动去用的，而是调用别人的方法需要才用)
- **另一个应用场景**
- ```
  import lombok.AllArgsConstructor;
  import lombok.Data;
  import lombok.NoArgsConstructor;
  @Data
  @NoArgsConstructor
  @AllArgsConstructor
  public class student { 学生类
  private String name;
  private int age;
  private int height;
  private String sex;
  }
  import java.util.Arrays; 导入数组包
  import java.util.Comparator; 导入排序包
  public class test3 { 主类
  public static void main(String[] args) {
  student[] s = new student[6];
  s[0] = new student("cxz1", 18, 180, "male");
  s[1] = new student("cxz2", 14, 183, "male");
  s[2] = new student("cxz3", 12, 181, "male");
  s[3] = new student("cxz4", 15, 159, "male");
  s[4] = new student("cxz5", 16, 160, "male");
  s[5] = new student("cxz6", 17, 190, "male");
  Arrays.sort(s, new Comparator(){ //声明一个比较器对象
  ```
  ```

```
@Override //查看源码发现这是一个接口，那么就可以用匿名内部类声明对象
public int compare(student o1, student o2) {
return o1.getAge() - o2.getAge(); //返回值是正数，升序排列
//返回值是负数，降序排列
}
});
for (int i = 0; i < s.length; i++) {
System.out.println(s[i]);
}
}
}
```

3. **函数式编程**
   - 类似于数学中的函数，输入的值一样，输出的值一样
   - **lambda表达式**：JAVA中的函数
   - 解决了什么问题：使用lambda表达式，可以替代匿名内部类，让程序代码更简洁
   - **语法格式：(被重写方法的形参列表)->{方法体}**
   - **lambda只能简化函数式接口的匿名内部类**
   - 函数式接口：有且仅有一个抽象方法的接口
   - ```
     public class test {
     public static void main(String[] args) {
     animal a = new animal(){ 匿名内部类
     @Override
     public void cry() {
     System.out.println("cry");
     }
     };
     a.cry();
     animal a1 = ()-> {System.out.println("cry");}; lambda表达式替代了匿名内部类
     a1.cry();
     }被重写方法的形参列表是cry()的这个括号，然后在他后面加上->.饭后加上方法体，最后加个;在}外面，就是lambda表达式
     }
     @FunctionalInterface // 函数式接口的声明注解
     interface animal { //函数式接口
     void cry();
     }
     ```
   - 可以简化以前的那个比较器对象
   - ```
     Arrays.sort(s, (student o1, student o2)-> {
     return o1.getAge() - o2.getAge(); //返回值是正数，升序排列
     ```

```
//返回值是负数，降序排列
});
这里是把匿名内部类，即从new开始的，替换为这个(com.cxz.innerclass3.student o1, student o2)-> {
return o1.getAge() - o2.getAge();});
for (int i = 0; i < s.length; i++) {
System.out.println(s[i]);
}
}
```

- **lambda表达式的省略规则**：
  - 参数类型可以全部省略不写
  - ```
    Arrays.sort(s, (o1, o2)-> {
    return o1.getAge() - o2.getAge(); //返回值是正数，升序排列
    //返回值是负数，降序排列
    }); 这里省略了参数类型student，因为参数类型可以由编译器自动推导出来
    ```
  - 如果只有一个参数，参数类型省略的同时()也可以省略，但多个参数不能省略()
  - ```
    Arrays.sort(s, (o1, o2)-> {
    return o1.getAge() - o2.getAge(); //返回值是正数，升序排列
    //返回值是负数，降序排列
    });这里就不能省略()，因为多个参数不能省略()，会报错的
    ```
  - 如果lambda表达式中只有一行代码，大括号可以不写，同时要省略分号;，如果这行代码是return语句，那么必须去掉return
  - ```
    Arrays.sort(s, (o1, o2)-> o1.getAge() - o2.getAge());
    这里省略了{}，同时省略分号;，同时省略return
    ```
- **方法引用**：
  - **静态方法的引用**：**类名::静态方法名**
  - 使用场景：如果某个lambda表达式里只是调用了一个静态方法，并且->前后参数形式一致，就可以使用静态方法引用
  - ```
    public class student {
    private String name;
    private int age;
    private int height;
    private String sex;
    public static int compare(student s1, student s2){ student类中的静态方法
    return s1.getAge() - s2.getAge();
    }
    public student(String name, int age, int height, String sex) {
    this.name = name;
    ```

```
this.age = age;
this.height = height;
this.sex = sex;
}
Arrays.sort(s, (s1, s2) -> s1.getAge() - s2.getAge());
Arrays.sort(s, (s1, s2) -> student.compare(s1, s2));
Arrays.sort(s, student::compare); 这里就是静态方法引用
因为满足了lambda表达式只调用了静态方法，并且->前后参数形式一致，都是s1,s2->s1,s2，所以可以引用静态方法进一步省略
```

- **实例方法的引用：对象名::实例方法名**
- 使用场景：如果某个lambda表达式里只是对象名调用了一个实例方法，并且->前后参数形式一致，就可以使用实例方法引用

```
public int comparebyheight(student s1, student s2){ student类中的实例方法
return s1.getHeight() - s2.getHeight();
}
import java.util.Arrays;
public class demo2 {
public static void main(String[] args) { 主类
student[] s = new student[6];
s[0] = new student("cxz1", 18, 182, "male");
s[1] = new student("cxz2", 18, 157, "male");
s[2] = new student("cxz3", 18, 156, "male");
s[3] = new student("cxz4", 18, 170, "male");
s[4] = new student("cxz5", 18, 160, "male");
s[5] = new student("cxz6", 18, 189, "male");
student t = new student(); 声明实例对象
Arrays.sort(s, (s1,s2)->t.comparebyheight(s1,s2));这是没有完全简化前的
Arrays.sort(s,t::comparebyheight); 实例方法引用，进一步简化上面的代码
for (int i = 0; i < s.length; i++) {
System.out.println(s[i].getHeight());
}
}
}
```

- **特定类型的方法引用：特定类的名称::方法**
- 使用场景：如果某个lambda表达式里只是调用了一个特定类型的实例方法，并且前面的参数列表中的第一个参数是作为方法的主调，后面的所有参数都是作为该实例方法入参的，那么就可以使用特定类型方法引用

```
import java.util.Arrays;
import java.util.Comparator;
public class demo3 {
```

```
public static void main(String[] args) {
//有一个字符串数组，里面有一些人的名字，请按照名字的首字母进行升序排序
String[] names = {"cxz", "Aws", "ffsg", "hfhtfh", "gfhdthr", "hthrds", "as"};
//把数组进行首字母排序Arrays.sort(names，comparator);
//忽略首字母大小进行排序，需要自己制定好
Arrays.sort(names,new Comparator(){ 匿名内部类
@Override
public int compare(String o1, String o2) {
return o1.compareToIgnoreCase(o2); //忽略大小写进行排序
调用特定类型的方法
}
});
//这是简化上面的匿名内部类
Arrays.sort(names,(o1, o2)-> o1.compareToIgnoreCase(o2));
接下来的是使用特定类型方法引用，最为简化
Arrays.sort(names,String::compareToIgnoreCase); //特定方法类型引用
//因为前面的参数列表的第一个参数是o1，是作为方法compareToIgnoreCase的主调
//后面的参数o2,是作为该实例方法入参的，故可以进行特定类型方法引用
//特定类型就是 String
for (String name : names) {
System.out.println(name);
}
}
}
```

- **构造器引用：类名::new**
- 应用场景：如果某个lambda表达式里只是在创建对象，并且->前后参数形式一致，就可以使用构造器引用。
- ```
  import lombok.AllArgsConstructor;
  import lombok.Data;
  import lombok.NoArgsConstructor;
  public class demo4 {
  public static void main(String[] args) { 主类
  carfactory factory = new carfactory() { 匿名内部类
  @Override
  public car getCar(String name) {
  return new car(name);
  }
  };
  carfactory factory1 = name -> new car(name); 经过lambda表达式简化
  carfactory factory2 = car::new; 构造器引用，进一步简化
  ```
  ```

```
car c1 = factory.getCar("BMW"); 创建对象
System.out.println(c1.getName()); 打印对象属性
}
}
@Data
@AllArgsConstructor
@NoArgsConstructor
class car{ 汽车类
private String name;
}
interface carfactory{ 汽车工厂接口
car getCar(String name);
}
```

4. **常用API**

- **String类**：String代表字符串，他的对象可以封装字符串数据，并提供很多方法完成对字符串的处理
  - 我们就需要创建字符串对象，封装字符串数据，并调用String类中的方法对字符串进行处理
  - **String字符串创建对象的方式：**
    - 方式一：**用""创建字符串对象**
    - ```
      String s1 = "hello";
      ```
    - 方式二：**调用String类的构造器初始化字符串对象**
    - ```
      public String() //创建一个空白字符串对象，不包含任何内容
      即String s2 = new String();
      ```
    - ```
      public String(String original) //使用指定的传入的字符串作为内容来创建字符串对象
      String s3 = new String("hello");
      ```
    - ```
      public String(char[] chars) //使用指定的字符数组来创建字符串对象
      即char[] chars= {'h','e','l','l','o'};
      String s4 = new String(chars);
      ```
    - ```
      public String(byte[] bytes) //使用指定的字节数组来创建字符串对象
      byte[] bytes = {97,98,99,100,101};
      String s5 = new String(bytes);
      ```
    - String创建对象的区别：以""形式写出的字符串对象，会存到字符串的常量池，且相同内容的字符串只存储一份。
    - ```
      String s1 = "hello";
      String s2 = "hello";s1，s2指向的是一个字符串对象。
      ```

    - 通过new方式创建字符串对象，每new一次都会产生一个新的对象放在堆内存中。
    - > String s3 = new String("hello");
      > String s4 = new String("hello");
      > s3，s4指向的是两个不同的对象，虽然内容一样，但是每次new都会创建一个不同的新对象。
  - **String类常用方法：**
  - > s1.**length()**; //获取字符串长度
  - > s1.**charAt(0)**; //获取指定索引位置的字符
  - > s1.**toCharArray()**; //将当前字符串转换成字符数组
  - > s1.**equals(s3)**; **//判断两个字符串的内容是否一样，一样返回true**
    > **判断字符串内容是否一样，不要用==，因为这默认比较的是地址，所以要用equals比较**
  - > s1.**equalsIgnoreCase(s3)**; //忽略大小写判断两个字符串的内容是否一样，一样返回true
  - > s1.**substring(1,3)**; //根据开始和结束索引进行截取，得到新的字符串，左闭右开
  - > s1.**substring(1)**; //截取从传入的索引1开始，一直截取到字符串末尾，得到新的字符串
  - > s1.**replace('l','o')**; //替换，将所有匹配的旧字符替换为新字符，得到新的字符串
  - > s1.**contains("ll")**; //判断当前字符串中是否包含传入的子串，包含返回true
  - > s1.**startsWith("he")**; //判断当前字符串是否以指定字符串开头，是返回true
  - > s1.**split("l")**; //将当前字符串按照传入的分隔符进行切割，得到字符串数组
- **ArrayList类**
  - 什么是集合：是一种容器，用来装数据，类似于数组(大小不能变)，但是集合大小可变，功能丰富。
  - 需要创建ArrayList对象，代表一个集合容器，调用ArrayList类提供的方法，对容器中的数据进行增删改查。
  - **创建ArrayList对象**
  - > **ArrayList< String > list = new ArrayList<>();**
  - **ArrayList类常用方法：**
  - > list.**add("hello")**; //添加元素，System.out.println(list.add("world")); 如果添加成功返回true，否则返回false
  - > list.**add(1,"world")**; //指定索引位置添加元素
  - > list.**remove(0)**; //删除指定索引位置的元素
  - > list.**remove("hello")**; //删除指定元素,System.out.println(list.remove("world"));如果删除成功返回true，否则返回false

- list.**get(0)**; //获取指定索引位置的元素
- list.**set(0,"world")**; //修改指定索引位置的元素
- list.**size()**; //获取集合大小