

一、实验名称

第4次上机-第7、8章内容

二、实验内容

page 60 第7章 基础训练1: 字符串的比较

目标: 理解 `==` 和 `equals` 方法的使用差异性

page 61 第7章 基础训练2: 字符串数据的提取

目标: 熟悉字符串的字符提取方法的应用

page 62 第7章 基础训练3: 字符串参数传递

目标: 理解 `String` 类型参数和 `StringBuffer` 类型参数的数据变化规律

page 63 第7章 基础训练4: 基本数据类型包装类的使用

目标: 理解包装类的作用

page 65 第7章 编程练习1: 输入一段英文句子, 将每个单词的首字母换为大写字母。例如, "I am very glad to see you" 的转换结果为 "I Am Very Glad To See You"

page 66 第7章 编程练习6:

从X星截获一份电码, 是如下一些数字串: 13、1113、3113、132113、1113122113、. YY博士经彻夜研究, 发现了如下规律: 第一个数随便是什么, 以后每个数都是对上一个数“读出来”。例如第2个数是对第1个数的描述, 意思是: 1个1, 1个3, 所以是: 1113。第3个数字意思是: 3个1, 1个3, 所以是: 3113。请编写一个程序, 可以从初始数字开始, 连续进行这样的变换。输入数据时, 第一行输入一个数字组成的串不超过100位。第二行输入一个数字n, 表示需要连续变换多少次, n不超过20。输出一个串, 表示最后一次变换完的结果。例如, 用户输入5和7, 则程序应该输出: 13211321322115。

page 70 第8章 基础训练2: 接口的定义与使用

目标: 理解一个类实现接口要在类中重现接口中定义的行为方法, 以及通过接口类型的引用实现接口的对象

page 71 第8章 基础训练3: 内嵌类的定义与使用

目标: 理解内嵌类的特点

三、实验环境

调试工具: Visual Studio Code Version: 1.98.0 Commit:

6609ac3d66f4eade5cf376d1cb76f13985724bcb OS: Ubuntu 24.04.1 LTS

服务配置: openjdk 21.0.6 2025-01-21 OpenJDK Runtime Environment (build 21.0.6+7-Ubuntu-124.04.1) OpenJDK 64-Bit Server VM (build 21.0.6+7-Ubuntu-124.04.1, mixed mode, sharing)

四、实验设计

page 60-63 第7章 基础训练1

(1) 字符串的比较

```
public class StringTest1 {  
    public static void main(String[] args) {  
        String x = "abc";  
        String y = "abc";  
        String z = new String("abc");  
        System.out.println(x == y);  
        System.out.println(x == z);  
        System.out.println(x.equals(z));  
        System.out.println(x.equals(y));  
    }  
}
```

以上代码的输出结果为

```
true  
false  
true  
true
```

可以初步得出结论: `==` 比较的是引用的地址值, 而 `equals` 比较的是字符串本身是否相等.

(2) 字符串大小的比较

```
public class StringTest2 {  
    public static void main(String[] args) {  
        String x = "abc1";  
        String y = "abc2";  
        String z = new String("ABC1");  
        System.out.println(x.compareTo(y));  
        System.out.println(x.compareTo(z));  
        System.out.println(x.equals(z));  
        System.out.println(x.equalsIgnoreCase(z));  
    }  
}
```

以上代码的输出结果为

```
-1  
32  
false  
true
```

总结用法差异:

方法	作用	区分大小写	返回值类型	返回值
<code>compareTo()</code>	按字典顺序（Unicode 值）比较两个字符串。	是	<code>int</code>	小于 0 (前者小), 0 (相等), 大于 0 (前者大)
<code>equals()</code>	比较两个字符串的内容是否完全相同。	是	<code>boolean</code>	<code>true</code> (完全相同), <code>false</code> (不同)
<code>equalsIgnoreCase()</code>	比较两个字符串的内容是否相同（忽略大小写）。	否	<code>boolean</code>	<code>true</code> (忽略大小写后相同), <code>false</code> (忽略大小写后仍不同)

page 61 第7章 基础训练2

(1) 在字符串中提取某个位置字符

```
public class StringTest3 {
    public static void main(String[] args) {
        String x = "a good idea";
        String subx = x.substring(0,1);
        System.out.println(x.charAt(0));
        System.out.println(subx);
    }
}
```

以上代码输出结果为:

```
a
a
```

可以看出 `x.substring(0,1)` 与 `x.charAt(0)` 的输出结果均为 `a`

方法	<code>charAt(int index)</code>	<code>substring(int beginIndex, int endIndex)</code>
函数参数	<code>int index</code> : 要获取的字符的索引位置。索引从 0 开始, 必须在 <code>0</code> 到 <code>string.length() - 1</code> 的范围内。	<code>int beginIndex</code> : 子字符串的起始索引 (包含)。必须大于等于 0。 <code>int endIndex</code> : 子字符串的结束索引 (不包含)。必须大于等于 <code>beginIndex</code> , 且小于等于 <code>string.length()</code> 。
核心功能	获取指定位置的单个字符	获取指定范围的子字符串
返回类型	<code>char</code>	<code>String</code>
性能	非常高	相对较低
适用场景	字符级别处理	字符串级别处理

(2) 将程序改为查找串中是否含有a字符

```
public class StringTest3 {
    public static void main(String[] args) {
        String x = "a good idea";
        // String subx = x.substring(0,1);
        // System.out.println(x.charAt(0));
        // System.out.println(subx);
        System.out.println("-----indexOf-----");
        if (x.indexOf("a") != -1) {
            System.out.println("有 a");
        } else {
            System.out.println("没有 a");
        }

        System.out.println("-----indexOf另一个调用写法-----");
        if (x.indexOf("a", 0) != -1) { // 指定从第几个字符开始找，从第一个开始找就相当于上
面的写法
            System.out.println("有 a");
        } else {
            System.out.println("没有 a");
        }
        System.out.println("-----startsWith-----");
        if (x.startsWith("a")) {
            System.out.println("以 a 开头");
        } else {
            System.out.println("不以 a 开头");
        }
    }
}
```

以上程序输出结果为:

```
-----indexOf-----
有 a
-----indexOf另一个调用写法-----
有 a
-----startsWith-----
以 a 开头
```

思考:

1. 除了书上给的 `indexOf(char character)` 还有 `indexOf(char character, int index)` 这样的调用方法, 可以从索引 `index` 开始, 查找 `character` 字符的下标.
2. 判断某字符是否以字符 `a` 开头只要修改程序为: `if (x.startsWith("a"))`

page 62 第7章 基础训练3

(1) 方法参数为String类型情形

程序输出结果为:

```
good bye!  
good bye
```

思考:

- 在 `method` 方法的执行过程中, 参数 `a` 的引用对象变为了另外一个字符串
- 没有影响, 因为在调用了 `method` 方法之后再次输出 `x` 依旧是 `good bye`, 并没有像在方法中添加的 `!`。

总结字符串类型参数传递的特点:

- **按值传递引用**: 方法传递的是对象的引用（实参的地址的拷贝），而不是对象本身。
- **不可变性**: 字符串是不可变的。任何对字符串的操作（如连接、替换等）都会创建一个新的字符串对象，原始字符串对象的内容不会被改变。
- **形参改变并不影响实参**: 由于按值传递引用，且字符串是不可变的，所以对方法内部形参字符串引用的任何修改（指向新的字符串对象）都不会影响到方法外部实参字符串的引用（仍然指向原来的字符串对象）。//TODO: String图

(2) 方法参数为 StringBuffer 类型情形

程序输出结果为:

```
good bye!  
good bye!
```

和之前 `String` 类型不一样, 调用 `method` 和之后再次输出 `x` 的结果相同, 说明在 `method` 中对 `x` 引用的对象本身进行了修改, 所以二者相同 //TODO: StringBuffer图

page 63 第7章 基础训练4

```

public class TestInteger {
    public static void main(String[] args) {
        Integer x = new Integer("123");
        Integer y = 12;
        Integer z = Integer.valueOf("1");
        System.out.println(x);
        System.out.println(y);
        System.out.println(z);
        int a = Integer.parseInt("888");
        System.out.println(a);
        System.out.println(Integer.toBinaryString(a));
        System.out.println(Integer.toString(a, 8));
        System.out.println(Integer.toString(a, 16));
    }
}

```

该程序输出结果为:

```

TestInteger.java:7: warning: [removal] Integer(String) in Integer has been deprecated
and marked for removal
    Integer x = new Integer("123");
                      ^
1 warning
123
12
1
888
1101111000
1570
378

```

包装类的作用主要有以下三点:

1. **将基本数据类型转换为对象:** 使得基本数据类型可以参与面向对象编程, 并存储在集合类中。
2. **提供实用方法:** 提供了各种类型转换、数值操作等相关的工具方法。
3. **支持自动装箱/拆箱:** 简化了基本数据类型和包装类对象之间的转换。

page 65 第7章 编程练习1

实现代码如下:

```

public class SwitchCap {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入英文句子: ");
        String orig = sc.nextLine();

        String[] words = orig.split(" ");
        for (int i = 0; i < words.length; i++) {
            if (words[i].length() > 0) {
                words[i] = words[i].substring(0,1).toUpperCase() +
words[i].substring(1).toLowerCase();
            }
        }

        String res = String.join(" ", words);
        System.out.println("转换后的句子: " + res);
        sc.close();
    }
}

```

测试结果如下:

```

请输入英文句子:
I am very glad to see you
转换后的句子: I Am Very Glad To See You

```

page 66 第7章 编程练习6

实现代码如下:

```

import java.util.Scanner;

public class C7P66E6 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String startNum = sc.nextLine();
        int n = sc.nextInt();

        String curNum = startNum;

        // 进行 n 次变换
        for (int i = 0; i < n; i++) {
            curNum = transform(curNum);
        }

        System.out.println(curNum);
        sc.close();
    }

    // 对前一个数字 "念出来"
    public static String transform(String num) {
        StringBuffer sb = new StringBuffer();
        int cnt = 1;
        for (int i = 0; i < num.length(); i++) {
            if (i + 1 < num.length() && num.charAt(i) == num.charAt(i + 1)) {
                cnt++; // 如果当前数字和下一个数字相同, cnt加1, 也就是`有cnt个num[i]` 继续累加
            } else {
                sb.append(cnt).append(num.charAt(i)); // 有cnt个num[i]
                cnt = 1; // 重置cnt为1
            }
        }

        return sb.toString();
    }
}

```

该程序输出结果为:

```

5
7
13211321322115

```

page 70 第8章 基础训练2

(1) 定义和实现接口

```
interface Listener {
    void action();
}

public class BusDriver implements Listener{
    public static void main(String[] args) {
        new BusDriver();
    }
}
```

尝试编译程序报错如下:

```
BusDriver.java:7: error: BusDriver is not abstract and does not override abstract
method action() in Listener
public class BusDriver implements Listener{
      ^
1 error
error: compilation failed
```

报错原因:

`BusDriver` 类实现了 `Listener` 接口, 但没有实现接口中定义的 `action()` 方法。

Java中一个类实现一个接口时, 它必须提供接口中所有抽象方法的具体实现。`Listener` 接口中定义了一个抽象方法 `action()`, 因此 `BusDriver` 类必须实现这个方法。

(2) 将 `BusDriver` 类中代码修改如下

```
interface Listener {
    void action();
}

public class BusDriver implements Listener{
    public void action() {
        System.out.println("看红绿灯!");
    }
    public static void main(String[] args) {
        BusDriver x = new BusDriver();
        x.action();
    }
}
```

编译运行成功输出 看红绿灯!

如果将 `public` 删去, 或者 加上 `private` 会得到以下的错误提示:

Cannot reduce the visibility of the inherited method from Listener

也就是说, 在 Java 中, 子类或实现类中的方法不能降低父类或接口中方法的访问权限。例如, 如果接口中的方法是 `public`, 那么实现类中的方法也必须是 `public`, 而不能是 `private`。

(3) 在程序中另增加一个类 Teacher 实现 Listener 接口

```
interface Listener {
    void action();
}

class Teacher implements Listener{
    public void action() {
        System.out.println("教书育人!");
    }
}

public class BusDriver implements Listener{
    public void action() {
        System.out.println("看红绿灯!");
    }
    public static void main(String[] args) {
        Listener x[] = {new BusDriver(), new Teacher()};
        x[0].action();
        x[1].action();
    }
}
```

程序正确输出.

```
看红绿灯!
教书育人!
```

在代码中尽管都通过 `Listener` 接口调用, 但由于实际对象类型不同, 运行时执行各自的 `action()` 方法: `BusDriver` 输出 "看红绿灯!", `Teacher` 输出 "教书育人!". 这是动态多态的体现: 同一个方法调用, 根据对象的实际类型, 产生不同的行为

page 71 第8章 基础训练3

(1) 两个并列类之间的对象成员数据访问

书中所给程序输出的结果是 100

`B` 类的方法访问了它所包含的 `A` 类的对象中的 `x` 成员变量, 而这个 `x` 成员变量在创建 `A` 的对象时被初始化为 100, 因此输出了 100。关键点在于 `B` 类的 `m()` 方法直接读取了同一

A 对象中的 x 变量的值。

(2) 将类B作为A的成员类

将程序改为书中要求修改的内容后, 输出结果依然为 100

对比二者特点: 以下是注释内容和非注释内容的比较表格, 简要分析它们的特点:

特性	独立	内嵌
类结构	包含两个独立的类: A 和 B, B 类通过构造函数依赖 A 类的实例。	包含一个外部类 A 和一个内部类 B, B 是 A 的非静态内部类, 直接访问 A 的成员。
依赖关系	B 类通过构造函数接收 A 类的实例, 显式依赖 A。	B 类直接访问外部类 A 的成员 (如 x), 隐式依赖 A。
访问权限	B 类需要通过 A 的实例访问 A 的成员 (如 a.x)。	B 类可以直接访问外部类 A 的成员 (如 x), 无需显式传递实例。
实例化方式	B 类的实例化需要显式传递 A 的实例: B b = new B(new A());。	B 类的实例化需要通过 A 的实例: A.B b = new A().new B();。

思考: 如果将类 A 中的成员变量 x、类 B 以及类 B 中的 m() 方法都加上 static 修饰符, 那么它们将变成静态成员。静态成员属于类本身, 而不是类的实例, 因此可以直接通过类名访问。

修改后的代码和调用方式如下:

```
public class A {  
    static int x = 100;  
  
    public static void main(String[] args) {  
        B.m();  
    }  
  
    static class B {  
        static void m() {  
            System.out.println(x);  
        }  
    }  
}
```

五、实验体会

通过本次实验，我学习了Java中的字符串处理、包装类、接口实现和内部类机制等核心概念，获得了以下收获：

字符串与包装类

- **字符串比较机制**：理解了 `==` 比较引用地址，而 `equals()` 比较内容，掌握了 `compareTo()` 和 `equalsIgnoreCase()` 的使用场景
- **字符串不可变性**：String对象一旦创建内容不可更改，任何修改操作都会创建新对象
- **字符提取方法**：掌握了 `charAt()` 和 `substring()` 的区别及适用场景
- **包装类功能**：学习了Integer等包装类将基本类型对象化、提供实用工具方法，以及支持自动装箱/拆箱的特性

接口与多态

- **接口实现规则**：实现类必须提供接口中所有抽象方法的具体实现，且不能降低访问权限
- **动态多态**：同一接口可有多个不同实现，通过接口引用调用时会根据实际对象类型执行相应方法
- **类型匹配**：理解了子类对象可以赋值给父类引用的向上转型机制

内部类机制

- **内部类访问特权**：非静态内部类可直接访问外部类的成员，无需显式引用
- **依赖关系**：非静态内部类依赖外部类实例创建，而静态内部类可独立存在
- **实例化方式**：掌握了不同类型内部类的创建语法，如 `外部类.内部类 变量 = 外部类实例.new 内部类()`

遇到了以下问题：

在解决page 66,编程题 （6）时，我最初采用了字符串拼接的方式，对 `StringBuffer` 类使用很不熟练。通过查阅文档学习相关方法我学习了 `StringBuffer` 类对字符串进行操作，确实比过去的方式要灵活强大。