

一、实验名称

第3次上机--5, 6章内容

二、实验内容

第5章：page 39, 基础训练1： 循序渐进了解对象知识

目标：了解对象和对象引用之间的关系

第5章: page 41, 基础训练2: 类成员与对象成员

目标：理解类成员和对象成员的访问特点

第5章: page 47, 编程题1: 编写一个代表三角形的类，其中三条边为三角形的属性并封装有求三角形的面积和周长的方法。分别用三条边为 3 4 5 和 7 8 9 的两个三角形进行测试

第6章: page 51, 基础训练1: 类的继承

目标：理解属性隐藏和方法覆盖的概念，熟悉对继承成员的访问规律

第6章: page 52, 基础训练2: 方法的参数多态

目标：了解方法调用的参数匹配处理

三、实验环境

调试工具:

Visual Studio Code

Version: 1.98.0

Commit: 6609ac3d66f4eade5cf376d1cb76f13985724bcb

OS: Windows_NT x64 10.0.26100

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

第5章：page 39, 基础训练1

(1) 构建Point类

```
public class Point {  
    int x, y;  
}
```

(2) 创建对象

使用系统提供的默认构造方式创建对象: `Point a = new Point();` 观察输出结果为:
`Lab03.Point@5f71c76a` , 确实为对象的引用地址

(3) 编写 `toString()` 方法

在 `public class Point` 类中添加了以下方法:

```
public String toString() {  
    return x + "," + y;  
}
```

重新编译运行后输出结果为: `0,0`

分析原因:

`System.out.println();` 方法会自动调用 `toString()` 方法, 然而我们在 `Point` 类中重写了 `toString()` 方法, 所以运行该java程序会运行我们刚刚重写过的 `toString()` 方法. 然而该方法中的 `x` 和 `y` 是成员变量并且没有被赋初值, 所以输出的就是默认值0, 最终结果是 `0,0`

(4) 让b和a引用同一个对象

在main方法中添加了如下代码:

```
Point b = a;  
a.x = 5;  
System.out.println(b);
```

重新编译运行之后输出为 `5,0` .

在输出 `b` 前我们更改的是 `a.x` 然而发现 `b.x` 同样被修改成了 5, 说明我们在创建 `b` 这个对象的代码中: `Point b = a;` 实际上是让二者引用了同一个对象, 所以对 `a` 中成员变量进行修改, 也会导致 `b` 中成员变量修改, 而且修改的本质上是同一个东西.

(5) 定义构造方法

在定义了一个有参构造方法之后, 由于在之前的代码中创建 `a` 对象使用了无参构造方法, 所以实际上还手动编写了一个无参构造方法

(6) 使用新定义的构造方法创建对象, 改变引用变量**b**, 让其指向新对象

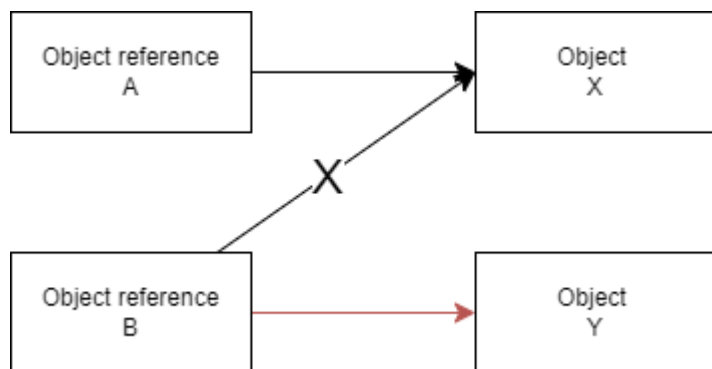
```
Point a = new Point();
Point b = a;
a.x = 5;

// 使用新定义的构造方法创建对象, 改变引用变量b, 使其指向新对象
b = new Point(8,3);
System.out.println(a);
System.out.println(b);
```

上述代码的输出结果如下:

```
5,0
8,3
```

对象和对象引用的关系: 对象是在内存中分配的具体实例, 包含实际的数据和状态; 而对象引用则是一个存储该对象内存地址的变量, 通过引用可以访问和操作所指向的对象。



图中在 `b = new Point(8,3);` 语句前是黑色箭头代表的状态, `a` 和 `b` 两个 **对象引用** 都指向了同一个 **对象** 所以他们的内容是相同的, 对其中一个的成员变量进行修改也会反应到另一个.

而执行了 `b = new Point(8,3);` 语句后, 就相当于把 `b` 指向 `x` 的箭头改成了指向 `y` 所以打印 `b` 的内容也就发生了改变

(7) 定义一个对象数组c

`Point c[] = {a, b};` 并输出 `c[0]` `c[1]` 得到结果如下:

```
5,0  
8,3
```

(8) 定义一个更大的数组

```
Point c[] = new Point[8];  
c[0] = a; c[1] = b;  
  
for (int k = 0; k < c.length; k++) {  
    System.out.println(c[k]);  
}
```

通过定义了一个更大的数组并循环输出, 结果如下

```
5,0  
8,3  
null  
null  
null  
null  
null  
null  
null
```

总结出对象数组元素的赋值特点如下:

- 引用复制而非对象复制：数组元素存储的是对象的引用（内存地址），复制的是引用，而不是实际的对象本身。
- 共享对象：多个数组元素可以引用同一个对象，对对象的修改会影响所有引用它的数组元素。
- 默认初始化为null：对象数组在创建时，未显式赋值的元素默认初始化为null。

(9) 添加c[6]的有参创建对象

在之前的循环输出 `c[]` 之前, 添加这段代码: `c[6] = new Point(4, 6);` 输出结果如下:

```
5,0
8,3
null
null
null
null
4,6
null
```

说明正常创建了一个对象, 并且对象引用 `c[6]` 正常指向了创建的对象

(10) 添加`c[7]`的无参创建对象

添加了这段代码: `c[7] = new Point();`, 编译输出报错如下:

```
Point.java:41: error: constructor Point in class Point cannot be applied to
given types;
    c[7] = new Point();
           ^
    required: int,int
    found:    no arguments
    reason: actual and formal argument lists differ in length
2 errors
error: compilation failed
```

因为我们之前写了一个 `Point` 类的有参构造函数, 而没有手动写无参构造函数, 所以尝试使用无参构造函数时发生了错误.

因此: 默认构造方法只会在没有手动编写任何构造方法时系统会自动提供

(11) 编写无参构造方法

添加了如下的无参构造方法:

```
public Point() {
    this(10, 10);
}
```

重新编译运行循环输出 `c[]` 得到结果如下

```
5,10
8,3
null
null
null
null
4,6
10,10
```

其中 `c[0]` 为 `5,10` 是因为其引用的是之前创建的对象 `a` , 其 `x` 变量经过了修改. `c[7]` 的值为 `10,10` 因为它是新创建的对象.

(12) 修改有参构造方法

```
public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
```

this的作用: `this`用于在类的方法中引用当前对象的成员变量或方法, 特别是在参数名与成员变量名冲突时, 明确指定操作的是当前对象的成员变量

思考: `toString()`中加上`this`

经过尝试 `toString()` 方法中的 `x` `y` 变量无论有无 `this` 都能正常输出

第5章: page 41, 基础训练2

(1) 没有static修饰的成员为对象成员

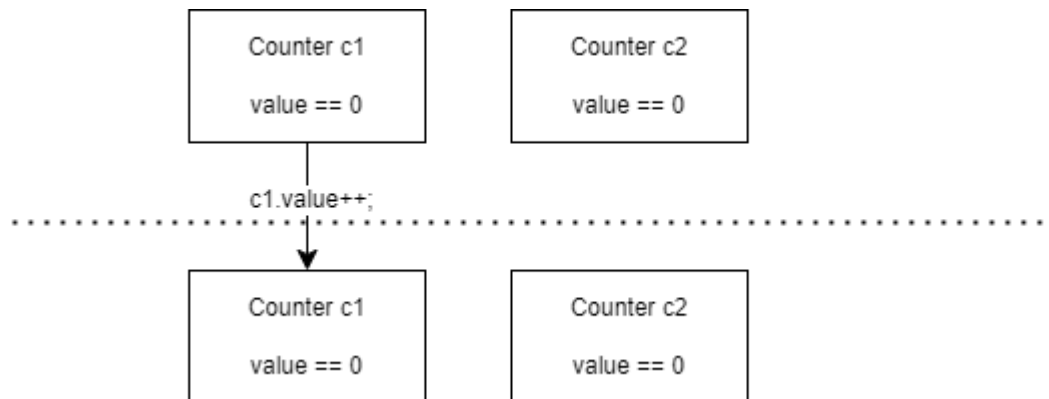
```
public class Counter {
    int value;

    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.value++;
        System.out.println(c1.value);
        System.out.println(c2.value);
    }
}
```

输出结果如下

```
1
0
```

因为分别 `new` 了两个不同的对象, 所以 `c1` 和 `c2` 分别指向两个不同的对象, 所以 `c1.value++;` 只会改变 `c1` 指向对象的变量, 而不会改变 `c2` 指向的对象的变量, 所以二者输出的结果不同.



(2) 在静态方法中能直接访问实例变量吗?

在 `main()` 方法中直接添加这条语句: `System.out.println(value);` 尝试编译运行报错如下:

```
Counter.java:14: error: non-static variable value cannot be referenced from a
static context
    System.out.println(value);
                        ^
1 error
error: compilation failed
```

报错原因: 非静态变量值不能从静态上下文中引用

(3) 将value属性改为static修饰

修改部分代码:

```
public class Counter {
    static int value;

    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        c1.value++;
        c2.value+=2;
        value+=3;
        System.out.println(c1.value);
        System.out.println(c2.value);
    }
}
```

输出结果如下:

```
6
6
```

解释: 由于 `value` 是 `static` 变量, 所有实例共享同一个 `value` . 无论通过哪个实例或类名修改 `value` , 都会影响所有实例的访问结果.

(4) 在实例方法中能访问类成员吗?

在 `Counter` 类中添加了一个实例方法:

```
public void incValue(int x) {
    value += x;
}
```

在 `main()` 方法中增加两行带代码:

```
incValue(5);
System.out.println(value);
```

尝试编译运行输出报错结果如下:

```
Counter.java:19: error: non-static method incValue(int) cannot be referenced
from a static context
    incValue(5);
    ^
1 error
error: compilation failed
```

原因: 非静态方法不能从静态上下文中引用

总结类成员和对象成员的差异

类成员（静态成员，`static`）：

- 属于类本身，所有类的实例共享。
- 通过类名直接访问（例如：`Counter.value`）。
- 在类加载时初始化。
- 静态方法只能访问静态成员。
- 常用于定义工具方法、常量。

对象成员（实例成员）：

- 属于类的每个实例（对象）。
- 通过对象名访问（例如：`c1.value`）。
- 在对象创建时初始化。
- 实例方法可以访问实例成员和静态成员。
- 常用于定义对象的状态和行为。

核心区别：

类成员是共享的，属于类；对象成员是独立的，属于每个对象。静态方法不能直接访问实例成员，但实例方法既能访问实例成员也能访问静态成员。访问方式的差别直接体现了他们的归属。

第5章: page 47, 编程题1

根据题目要求编写代码如下:

```

import java.util.Scanner;

public class Triangle {
    double a, b, c;

    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    double getLength() {
        return a + b + c;
    }

    double getArea() {
        double s = getLength() / 2;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    boolean isValied() {
        return a + b > c && a + c > b && b + c > a;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("请分别输入三角形的三条边，用空格隔开:");
        double a = sc.nextDouble();
        double b = sc.nextDouble();
        double c = sc.nextDouble();
        Triangle t = new Triangle(a, b, c);
        if (t.isValied()) {
            System.out.println("周长: " + t.getLength());
            System.out.println("面积: " + t.getArea());
        } else {
            System.out.println("输入的三条边无法构成三角形");
        }
        sc.close();
    }
}

```

使用 3 4 5 和 7 8 9 作为测试样例得到结果如下:

```
> java Triangle.java
请分别输入三角形的三条边，用空格隔开：
3 4 5
周长：12.0
面积：6.0
> java Triangle.java
请分别输入三角形的三条边，用空格隔开：
7 8 9
周长：24.0
面积：26.832815729997478
```

第6章: page 51, 基础训练1

(1) 继承关系中的覆盖与重载问题

```
class Parent {
    int x = 100;

    void m() {
        System.out.println(x);
    }
}
public class Child extends Parent {
    int x = 200;

    public static void main(String[] args) {
        Child a = new Child();
        a.m();
        System.out.println(a.x);
    }
}
```

尝试编译运行以上程序, 输出结果如下:

```
100
200
```

- `a.m();` 调用了 `Parent` 类的 `m()` 方法, 而 `m()` 方法打印 `Parent` 类的 `x`, 即 `100`.
- `System.out.println(a.x);` 访问的是 `Child` 类的 `x`, 即 `200`.

(2) 在子类中覆盖父类方法

在子类中增加一个和 `Parent` 类中 `m()` 方法形态一样的方法:

```
void m() {  
    System.out.println("x="+x);  
}
```

输出结果为

```
x=200  
200
```

所以, 如果在子类中有一个方法与父类中方法一样, 那么就会优先调用子类的方法, 覆盖父类的方法.

(3) 通过super访问父类方法和属性

修改子类 `m()` 方法如下:

```
void m() {  
    System.out.println("x="+x);  
    System.out.println("super.x="+super.x);  
    super.m();  
}
```

输出如下:

```
x=200  
super.x=100  
100  
200
```

`super` 能够访问父类中的变量, 调用父类中的方法.

(4) 站在父类引用的角度看成员, 熟悉对继承成员访问的特点。

通过修改创建a对象的语句为: `Parent a = new Child();` 可以使得父类引用变量引用子类对象, 修改后输出结果如下:

```
x=200
super.x=100
100
100
```

显然最后一行输出的 `a.x` 内容是不同的，这是因为在这里创建的 `x` 引用类型引用的是 `Parent` 类创建的对象，其成员变量 `x==100`

而改动之前引用的是 `Child` 类创建的对象，其成员变量 `x==200`，因此输出结果不同

第6章: page 52, 基础训练2

(1) 参数多态的方法匹配

```

public class MethodMatch {
    int x = 200;

    public MethodMatch() {
        x = 300;
    }

    public MethodMatch(int x1) {
        x = x1;
    }

    public void m1(int k) {
        x = x + k;
    }

    public void m1() {
        x = x - 1;
    }

    public void m1(int x, int y) {
        this.x = this.x + x * y;
    }

    public static void main(String[] args) {
        MethodMatch a = new MethodMatch();
        a.m1(2,3);
        MethodMatch b = new MethodMatch(20);
        b.m1(50);
        a.m1(9);
        System.out.println("a.x="+a.x);
        System.out.println("b.x="+b.x);
    }
}

```

以上程序输出如下：

```

a.x=315
b.x=70

```

(2) 修改参数匹配

通过把 `main` 方法中的 `b.m1(50)` 改为了 `b.m1(50.2)` 尝试编译运行报错如下：

```
MethodMatch.java:30: error: no suitable method found for m1(double)
    b.m1(50.2);
      ^
    method MethodMatch.m1(int) is not applicable
      (argument mismatch; possible lossy conversion from double to int)
    method MethodMatch.m1() is not applicable
      (actual and formal argument lists differ in length)
    method MethodMatch.m1(int,int) is not applicable
      (actual and formal argument lists differ in length)
1 error
error: compilation failed
```

提示没有对应 `m1(double)` 相符的方法, 这是因为我们没有在编写方法的时候没有编写 `m1(double)` 形式的多态

总结出参数转换匹配的规律如下：

- **参数类型匹配：**
 - 编译器会首先尝试找到参数类型完全匹配的方法。
 - 如果没有完全匹配的方法，编译器会尝试进行类型转换（如从 `int` 转换为 `double`）。
- **参数数量匹配:**

编译器会根据传递的参数数量来选择合适的方法。如果参数数量不匹配，编译器会报错。

- **优先级顺序：**
 - 精确匹配优先：编译器优先选择参数类型和数量完全匹配的方法。
 - 自动类型转换：如果没有精确匹配的方法，编译器会尝试进行自动类型转换（如从 `int` 转换为 `double`）。

五、实验体会

通过本次实验，我学到了以下内容：

- **对象与引用机制**

通过Point类的实验，理解了对象是内存中的实体，而引用只是指向对象的指针。当多个引用指向同一对象时，通过任何引用的修改都会影响该对象的状态。这解释了为什么 `Point b = a` 只复制引用而不复制对象本身。

- **类成员与对象成员** Counter类实验明确展示了 `static` 修饰的类成员与实例成员的区别：

- 类成员由所有实例共享，可通过类名直接访问
- 对象成员属于各个独立的实例，需通过对象引用访问
- 静态方法不能访问实例成员，但实例方法可以访问所有成员

• 继承与多态

Parent-Child类的实验展示了Java继承机制中的方法覆盖和变量隐藏特性：

- 方法调用遵循动态绑定，取决于对象的实际类型
- 变量访问遵循静态绑定，取决于引用的声明类型
- 通过super关键字可访问父类的成员

• 数组与参数传递

实验中明确了不同类型数组的默认初始化值（整型为0，引用类型为null），以及参数传递机制：

- 基本类型传递值的副本，方法内的修改不影响原变量
- 引用类型传递引用的副本，方法内的修改会影响原对象

实验中遇到的问题：

在尝试编译运行 `Child.java` 时，终端报错如下

```
> java Child.java
error: can't find main(String[]) method in class: Lab03.Parent
```

解决方法：

检查目录结构，我的 `Child.java` 包含在 `Lab03` 包中。应该在 `Lab03` 的父目录下执行编译命令。我的文件结构是 `src/Lab03/Child.java` 所以cd到 `src` 目录下，执行 `javac Lab03/Child.java`，然后使用 `java Lab03.Child` 运行。得到了正确的结果。