



主讲教师 张 智
计算机学院软件工程系
课程群: 421694618

4 类和对象

4.1 [类和对象](#)

4.2 [构造函数](#)

4.3 [方法重载](#)

4.4 [this关键字](#)

4.5 [static关键字](#)

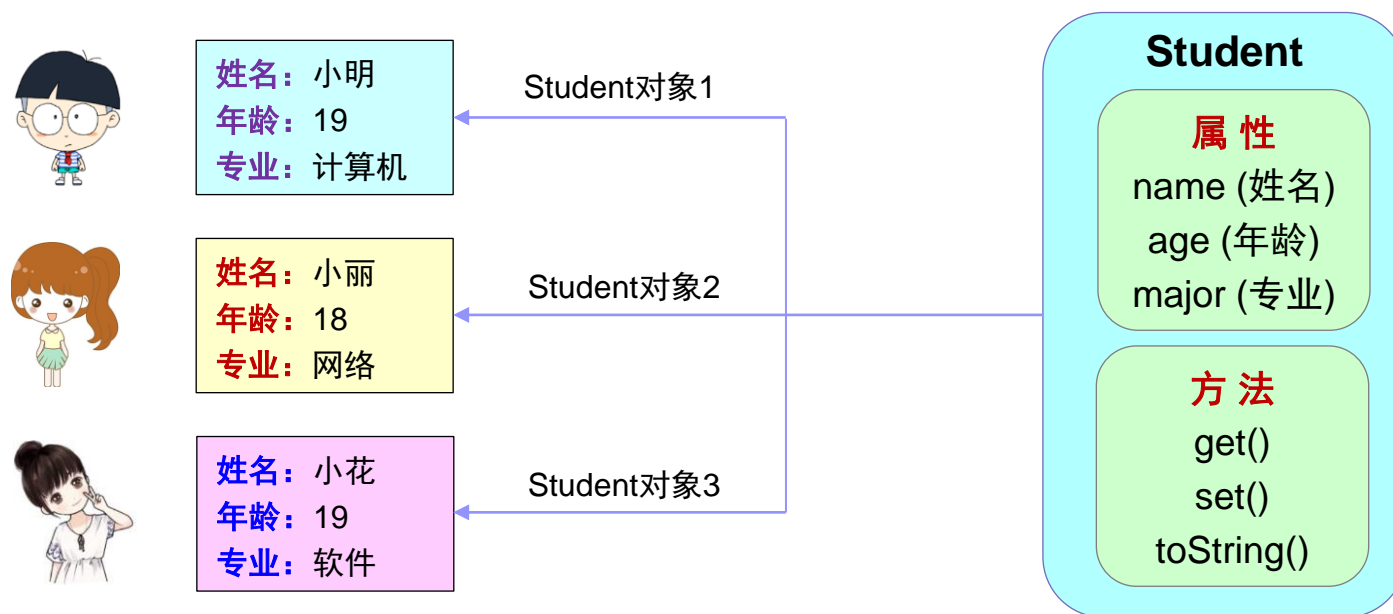
4.6 [final关键字](#)

4.7 [编程练习](#)

【附录1】[Date类和Calendar类](#)

4.1 类和对象

- **类**：是一组^①属性(成员变量/常量)以及^②属性上的方法(函数/操作)的^③封装体
- **对象**：是类的实例(具体)，类是对象的模板(抽象)



面向对象三大核心特性：

- **封装**
- **继承**
- **多态**

类的声明

```
package edu.wust.examples; //打包
```

```
public class Student {  
    private String name;  
    private int age;  
    private String major;
```

属性 一般为private

```
    public String getName() {  
        return name;  
    }
```

```
    public void setName(String name) {  
        this.name = name;  
    }
```

this表示当前对象

```
    public int getAge() {  
        return age;  
    }
```

方法

一般为public

```
    public void setAge(int age) {  
        this.age = age;  
    }
```

```
    public String getMajor() {  
        return major;  
    }
```

```
    public void setMajor(String major) {  
        this.major = major;  
    }
```

@Override

```
    public String toString() {  
        return "我叫" + name + "," + age + "岁,专业是" + major;  
    }
```

Student.java

封装体

私有属性的好处

- 防止对封装数据的未授权访问
- 有助于保证数据完整性
- 当类的变量必须改变时，可以限制发生在整个应用程序中的“连锁反应”

对象的创建和成员访问

```
public class Test {  
    public static void main(String args[]) {  
        Student s = new Student(); ← 创建对象(实例化)  
        s.setName("小花");  
        s.setAge(19);  
        s.setMajor("软件");  
        System.out.println( s );  
    }  
}
```

通过对象.方法()来访问公有成员方法

输出对象时, 将自动调用toString()

运行结果

我叫小花,19岁,专业是软件

思考: 如果没有调用set, 则结果如何

我叫null,0岁,专业是null

注: 系统会使用默认值初始化成员变量

了解：关于Java Bean的概念

- 对于遵循以下规范的Java类就称为Java Bean：
 - JavaBean是一个公共的类（public class）
 - JavaBean有一个不带参数的构造方法（或默认构造函数）
 - JavaBean通过setXxx方法设置属性，通过getXxx方法获取属性

Java Bean在Java EE开发中，通常用于封装数据，是一种可重复使用的，跨平台的软件组件

[【返回】](#)

4.2 构造函数

- 新对象的创建和初始化通常是调用一个构造函数的来实现的。
 - 例如：Student s = new Student(); 对象s由Student()这个构造函数创建
- 每个类至少要有一个构造函数，如果没有编写构造函数，系统会提供一个默认构造函数。
 - 默认构造函数：不带参数，函数体是空的，形如：Student() { }
- 可自定义构造函数，并向构造函数传递创建对象所需参数。

注：一旦自定义构造函数，系统默认的构造函数将自动关闭。如果要使用默认构造函数，则必须显示声明。

自定义构造函数

好处：(1) 给初始化带来多种形式；(2) 为用户提供更大的灵活性

```
public class Student {
```

```
...
```

```
public Student(String name, int age, String major) {
```

```
    this.name = name;
```

```
    this.age = age;
```

```
    this.major = major;
```

```
}
```

```
...
```

```
}
```

自定义构造函数

说明

- 构造函数的名称必须与类名相同
- 构造函数无任何返回值
- 构造函数不指定类型(包括void)
- 构造函数前面一般加public (也可用private)
- 一个类可以有多个构造函数(重载, 见下节)

构造函数使用

```
public class Test {  
    public static void main(String[] args) {  
        // Student s = new Student(); //报错：默认构造函数自动关闭  
        Student s1 = new Student("小花", 19, "软件");  
        System.out.println( s1 );  
    }  
}
```

运行结果

我叫小花,19岁,专业是软件

补充：匿名对象

- 匿名对象就是没有明确的给出名字的对象
- 如果一个对象只用一次，就可以使用匿名对象
- 例如： `System.out.println(new Student("小花", 19, "软件"));`

匿名对象

了解：Java没有析构函数

- Java没有析构函数，代之以**垃圾回收机制**(garbage collection,简称GC)，但垃圾回收并不等同于“析构”。
- 垃圾回收是Java虚拟机(**JVM**)提供的一种用于在**空闲时间不定时回收**无用对象所占据内存空间的一种机制。
- 如果想提示垃圾回收器尽快执行垃圾回收操作，可在程序中使用：
 - **System.gc();** //注：该命令不保证会确实进行垃圾回收

[【返回】](#)

4.3 方法重载 (overload)

```
public void indexOfInt(int ch) { ... }
```

```
public void indexOfString(String str) { ... }
```

```
public void indexOfFromIndex1(int ch, int fromIndex) { ... }
```

```
public void indexOfFromIndex2(String str, int fromIndex) { ... }
```

背景：功能相似，但函数名不同会造成用户调用繁琐

用重载处理

```
public int indexOf(int ch) { ... }
```

```
public int indexOf(String str) { ... }
```

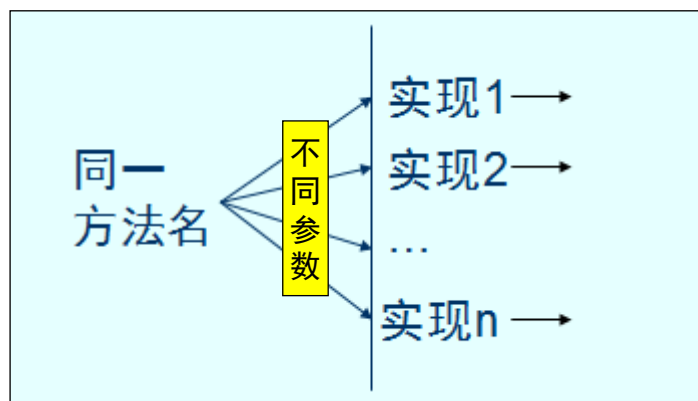
```
public int indexOf(int ch, int fromIndex) { ... }
```

```
public int indexOf(String str, int fromIndex) { ... }
```

重载好处：功能类似的方法使用同一名字，更容易记住，调用起来更简单

重载概念

- 方法重载：方法名相同，但各自的参数不同 (参数类型、参数个数、参数顺序至少有一项不同)
- 系统在编译时能够根据参数情况自动选择一个合适的方法



重载的条件 (关键在参数)

- 方法名必须相同
- 方法的参数类型、参数个数、参数顺序至少有一项不相同

注意：

- 不以返回类型的不同或方法的修饰符不同来进行重载。

注意：方法重载的返回值类型和修饰符通常都是相同的

示例：

哪些可以和①重载？ 答案：②④

- ① `public int fun(int, double)`
- ② `public int fun(int)`
- ③ `private double fun(int, double)`
- ④ `int fun(double, int)`
- ⑤ `public int funs(int)`

程序阅读

```
public class Test {  
  
    public void foo() {  
        System.out.println("No parameters");  
    }  
  
    public void foo(int a) {  
        System.out.println("a: " + a);  
    }  
  
    public void foo(int a, int b) {  
        System.out.println("a and b: " + a + " " + b);  
    }  
}
```

```
public void foo(double a) {  
    System.out.println("double a: " + a * a);  
}  
  
public static void main(String[] args) {  
    Test t = new Test();  
    t.foo();  
    t.foo(2);  
    t.foo(2, 3);  
    t.foo(2.0);  
    t.foo(2L);  
    t.foo(2.0f);  
}
```

注意这2个

运行结果

```
No parameters  
a: 2  
a and b: 2 3  
double a: 4.0  
double a: 4.0  
double a: 4.0
```


构造函数重载

好处：(1) 给初始化带来多种形式；(2) 为用户提供更大的灵活性

定义3个构造函数

```
public class Student {  
    ...  
    public Student() { ← 显式开启默认的构造函数  
    }  
  
    public Student(String name, int age, String major) { ← 全参数构造函数  
        this.name = name;  
        this.age = age;  
        this.major = major;  
    }  
  
    public Student(String name, int age) { ← 部分参数构造函数  
        this.name = name;  
        this.age = age;  
        this.major = "待定";  
    }  
    ...  
}
```

构造函数使用

```
public class Test {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        Student s2 = new Student("小明", 19, "软件");  
        Student s3 = new Student("小花", 18);  
        System.out.println( s1 );  
        System.out.println( s2 );  
        System.out.println( s3 );  
    }  
}
```

我叫null,0岁,专业是null
我叫小明,19岁,专业是软件
我叫小花,18岁,专业是待定

[【返回】](#)

4.4 this关键字

■ this的3种用法:

■ this.属性名



■ 普通成员方法访问成员变量时无须使用 this 前缀，但如果方法里有局部变量和成员变量同名，则需要使用 this 关键字来访问类中的属性，以区分类的属性和方法中的参数。

■ this.方法名



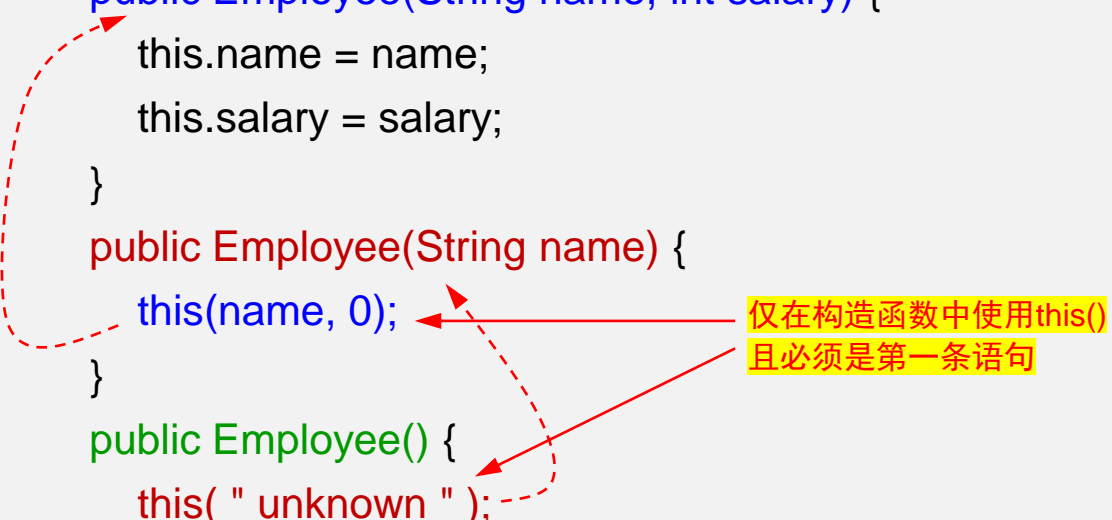
■ 一个方法访问该类中定义的其他方法时加不加 this 前缀的效果是完全一样的(注: static 修饰的方法中不能使用 this 引用)

■ this()访问构造方法

this()访问构造方法

- 一个类的构造函数之间可以相互调用
- 调用方法: **this([参数])**
- **特别注意:**
 - this() 仅在类的构造函数中使用, 别的地方不能用
 - this() 必须是整个构造函数的第一个可执行语句

```
public class Employee {  
    private String name;  
    private int salary;  
    public Employee(String name, int salary) {  
        this.name = name;  
        this.salary = salary;  
    }  
    public Employee(String name) {  
        this(name, 0);  
    }  
    public Employee() {  
        this( " unknown " );  
    }  
    @Override  
    public String toString() {  
        return "Employee{name='" + name + "', salary='" + salary + "'}";  
    }  
}
```



Employee e1=new Employee("zz",100);

则 e1.name="zz" e1.salary=100

Employee e2=new Employee("qq");

则 e2.name="qq" e2.salary=0

Employee e3=new Employee();

则 e3.name="unknown" e3.salary=0

[【返回】](#)

4.5 static关键字

- 通常情况下，类成员必须通过类的对象来访问，但是可以创建这样的成员，它的使用**完全独立于该类的任何对象**。
- **static关键字**用来**声明成员属于类**，**被类的所有实例共享**，**而不依赖于类的特定实例**。
- 如果一个成员被声明为static，它就能够在它的类的任何对象创建之前被访问，而不必引用任何对象（只要这个类被加载，JVM就可以根据类名找到它们）。

■ 调用静态成员的语法：**类名.静态成员**


static用法

- 静态变量
- 静态方法
- 静态代码块

【返回】

1. 静态变量

- 程序运行时，Java 虚拟机为静态变量只分配一次内存，在加载类的过程中完成静态变量的内存分配
- 在类的内部，可以在任何方法内直接访问静态变量
- 在其他类中，可以通过 类名.静态变量 来访问（也可通过对象，但不推荐）



注：静态变量通常加public修饰

静态变量作用

- 静态变量可被类的**所有实例共享**，因此静态变量可以作为实例之间的共享数据，增加实例之间的交互性。
- 如果类的所有实例都包含一个相同的常量属性，则可以把这个属性定义为静态常量类型，从而**节省内存空间**。

静态变量示例

```
public class StaticVar {  
    //定义静态变量  
    public static String logo = "wust";  
  
    public static void main(String[] args) {  
        // 类的内部可直接访问logo  
        System.out.println("第1次访问静态变量，结果为：" + logo);  
  
        // 通过类名访问logo  
        System.out.println("第2次访问静态变量，结果为：" + StaticVar.logo);  
  
        // 通过对象obj1访问logo  
        StaticVar obj1 = new StaticVar();  
        obj1.logo = obj1.logo + " cs!";    //静态变量被修改  
        System.out.println("第3次访问静态变量，结果为：" + obj1.logo);  
  
        // 通过对象obj2访问logo  
        StaticVar obj2 = new StaticVar();  
        System.out.println("第4次访问静态变量，结果为：" + obj2.logo);  
    }  
}
```

静态变量也可赋值

静态变量通常加public修饰

类的内部可直接访问

通过类名访问(推荐)

通过对象访问

通过对象访问

[【返回】](#)

2. 静态方法

注：实例方法可以直接访问所属类的静态变量、静态方法

■ 静态方法只能调用静态成员（static方法或static属性）

- 原因：静态方法在class装载时首先完成，比构造函数早，此时非静态属性和方法还没完成初始化，所以静态方法不能调用非静态方法和属性。

■ 静态方法不能以任何方式引用 this 或 super

- 原因：静态方法不需要通过它所属的类的任何实例就可以被调用，因此在静态方法中不能使用 this 关键字。
- 和this关键字一样，super关键字也与类的特定实例相关，所以在静态方法中也不能使用super关键字

静态方法示例

```
public class Test {  
    private int x;    静态变量  
    public static int y;  
    静态方法  
    public static void main(String args[]) {  
        x = 9;        // 报错 ← 静态方法不能访问非静态成员  
        this.y = 10 ; // 报错 ← 静态方法不能以任何方式引用this 或super  
        y = 10 ;      // ok 静态方法只能调用静态成员  
        Test.y = 10;  // ok 通过 类名.静态成员 访问  
    }  
}
```

静态方法示例

```
class Simple {
```

```
    //普通方法(实例方法)
```

```
    public void go1() {
```

```
        Test t = new Test();
```

```
        t.go3(); //ok
```

```
    }
```

对象能访问静态方法，但不推荐

```
    // 静态方法
```

```
    public static void go2() {
```

```
        System.out.println("Go2...");
```

```
    }
```

```
}
```

```
public class Test {
```

```
    // 静态方法
```

```
    public static void go3() {
```

```
        System.out.println("Go3...");
```

```
    }
```

```
    //普通方法(实例方法)
```

```
    public void go4() {
```

```
        System.out.println("Go4...");
```

```
    }
```

```
    // 静态方法
```

```
    public static void main(String[] args) {
```

```
        new Simple().go1(); //OK
```

```
        go3(); //OK
```

```
        go4(); //报错
```

静态方法不能访问非静态成员

```
        Simple.go2(); //OK
```

```
    }
```

```
}
```

推荐：类名.静态方法

匿名对象访问普通方法

静态方法访问静态成员

```
class Value {  
    public static int c = 0;  
    public static void inc() {  
        c++;  
    }  
}  
  
public class Count {  
    public static void prt(String s) {  
        System.out.println(s);  
    }  
    public static void main(String[] args) {  
        Value v1 = new Value();  
        Value v2 = new Value();  
        prt("v1.c=" + v1.c + " v2.c=" + v2.c);  
        Value.inc();    //也可 v1.inc(); 但不推荐  
        prt("v1.c=" + v1.c + " v2.c=" + v2.c);  
    }  
}
```

运行结果

v1.c=0	v2.c=0
v1.c=1	v2.c=1

[【返回】](#)

3. 静态代码块

```
static {  
    语句块;  
}
```

- 静态代码块可以置于类中的任何地方，类中可以有多多个静态初始化块
- 静态代码块只会执行一次，且在类被第一次装载时
- 静态代码块按它们在类中出现的顺序被执行

通常将一次性的初始化操作放在静态代码块中进行

静态代码块示例


```
class StaticCode {  
    public static int i = 5; ①  
    static { ← 静态代码块在类被加载时只会执行一次  
        System.out.println("Static code i=" + i++ );  
    }  
}  
public class Test {  
    public static void main(String args[]) {  
        System.out.println("Main code: i=" + StaticCode.i );  
    }  
}
```

StaticCode类第一次
加载时会先执行static

程序阅读

```
class StaticCode {
```


```
    public static int count = 0; ①
```

④ {  **非静态代码块(不推荐用)**
在创建对象时会自动执行

```
        count++;
```

```
        System.out.println("非静态代码块 count=" + count);
```

```
    }
```

② **static** {  **静态代码块在类被加载时只会执行一次**

```
    count++;
```

```
    System.out.println("静态代码块1 count=" + count);
```

```
 }
```

③ **static** {

```
    count++;
```

```
    System.out.println("静态代码块2 count=" + count);
```

```
 }
```

```
}
```

```
public class Test {
```

```
    public static void main(String args[]) {
```

```
        StaticCode sc1 = new StaticCode();
```

```
        StaticCode sc2 = new StaticCode();
```

```
    }
```

```
}
```

运行结果

```
静态代码块1 count=1
静态代码块2 count=2
非静态代码块 count=3
非静态代码块 count=4
```

[【返回】](#)

4.6 final关键字

- final类
- final方法
- final变量
- final参数

【返回】

1. final类

- 在设计类时候，如果这个类的实现细节不允许改变，类不需要有子类，并且确信这个类不会被扩展，那么就设计为final类

```
public final class 类名 {  
    ...  
}
```

← final类，不能被继承

- final类不能被继承。因此final类的成员方法没有机会被覆盖
 - 如String、Math等类都是 final 类

[【返回】](#)

2. final方法

■ final方法可以被继承，但不能被重写(覆盖)

```
public class 类名 {  
    public final void f2() { ← final方法，可以被继承，但不能被子类重写  
        System.out.println("f2");  
    }  
}
```

■ 使用final方法的原因有二：

- 把方法锁定，防止任何子类修改它的实现
- 高效。编译器在遇到调用final方法时候会转入内嵌机制，大大提高执行效率

[【返回】](#)

3. final变量

称为空白final变量

- final 修饰基本类型变量时即成为常量，且只能赋值一次 (可先声明，不给初值，但在使用之前必须被初始化一次)

例如：

```
final double PI = 3.14159;    // 通常称为符号常量
```

```
PI = 3.14;    // 错，不能改
```

```
final String LOGO;           // 空白final变量
```

```
// LOGO = "wust";           // OK，先定义后初始化
```

```
System.out.println( LOGO );  // 报错，使用前还没有初始化
```

3. final变量（续）

- final 修饰引用类型变量时，只保证这个引用类型变量所引用的地址不会改变，但其自身的成员变量的值可以被改变

例如：

```
final int[] arr = { 5, 6, 12, 9 };
```

```
arr[2] = -8;    // OK，对数组元素赋值，合法
```

```
arr = null;    // 报错：arr重新赋值，非法
```

```
final Person p1 = new Person(45);
```

```
p1.setAge(23);    // OK，改变p1对象age值，合法
```

```
p1 = p2;    // 报错：p1重新赋值，非法
```

[【返回】](#)

4. final参数

- 当函数参数为final类型时，只可以读取使用该参数，但是无法改变该参数的值。

```
public class Test {  
    public static void main(String[] args) {  
        foo(2);  
    }  
    public static void foo( final int n ) {  
        n++;    // 报错，final类型值不允许改变  
        System.out.print(n);  
    }  
}
```

[【返回】](#)

4.7 编程练习

1、定义一个三角形Triangle类：

- 数据成员：a, b, c（double类型）；
- 具有的操作：
 - (1) 构造函数
 - (2) 判断是否构成三角形 isTriangle()
 - (3) 计算三角形面积 getArea()
 - (4) 显示三角形信息 toString()
 - (5) getter/setter（本例略）

Triangle.java

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

//判断是否构成三角形

```
public boolean isTriangle() {  
    return a + b > c && a + c > b && b + c > a;  
}
```

//求面积

```
public double getArea() {  
    double s = (a + b + c) / 2.0;  
    if ( isTriangle() )  
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));  
    else  
        return -1;  
}
```

↑
海伦公式

@Override

```
public String toString() {  
    return "a=" + a + ", b=" + b + ", c=" + c;  
}  
}
```

测试类：

Test.java

```
public class Test {  
    public static void main(String[] args) {  
        Triangle t1 = new Triangle();  
        System.out.println( t1 );  
        System.out.println( "面积="+t1.getArea() );  
  
        Triangle t2 = new Triangle(3,4,5);  
        System.out.println( t2 );  
        System.out.println( "面积="+t2.getArea() );  
  
        Triangle t3 = new Triangle(3);  
        System.out.println( t3 );  
        System.out.println( "面积="+t3.getArea() );  
    }  
}
```

```
a=0.0, b=0.0, c=0.0  
面积=-1.0  
a=3.0, b=4.0, c=5.0  
面积=6.0  
a=3.0, b=3.0, c=3.0  
面积=3.897114317029974
```

编程练习：

2、定义一个People类：

数据成员：name (String)、birthday ([Date](#))

具有的操作：

(1) 构造函数

(2) 重载方法：eat()、eat(s)

(3) birthday修改操作：getBirthday()、getAge() -- 计算年龄

说明：Date和Calendar类用法见附录1

■ Date和Calendar类：

```
import java.util.Calendar;
```

```
import java.util.Date;
```

■ 格式化Date数据的SimpleDateFormat类：

```
import java.text.SimpleDateFormat;
```

```

public class People {
    private String name;
    private Date birthday;

    public People() {
    }

    public People(String name, Date birthday) {
        this.name = name;
        this.birthday = birthday;
    }

    // eat() 两个重载方法
    public void eat() {
        System.out.println("People can eat");
    }

    public void eat(String food) {
        System.out.println("People can eat " + food);
    }

    public Date getBirthday() {
        return birthday;
    }

    /*
    * 有关Date和Calendar类用法见附录
    */
}

```

```

//计算年龄
public int getAge() {
    // 获取当前日期和时间
    Calendar now = Calendar.getInstance();
    // 处理birthday
    Calendar birth = Calendar.getInstance();
    birth.setTime(birthday);

    // 年龄 = 当前年 - 出生年
    int age = now.get(Calendar.YEAR) - birth.get(Calendar.YEAR);
    if (age <= 0) {
        return 0;
    }

    // 如果当前月份小于出生月份: age-1
    // 如果当前月份等于出生月份, 且当前日小于出生日: age-1
    int currMonth = now.get(Calendar.MONTH);
    int currDay = now.get(Calendar.DAY_OF_MONTH);
    int birthMonth = birth.get(Calendar.MONTH);
    int birthDay = birth.get(Calendar.DAY_OF_MONTH);
    if ((currMonth < birthMonth) || (currMonth == birthMonth && currDay <=
    birthDay)) {
        age--; //年龄减1
    }
    return age < 0 ? 0 : age;
}
}

```

Calendar实例化方法
不能使用new

将Date转为Calendar类型

Calendar的get()方法获取日期相关参数
具体见附录说明

测试类

SimpleDateFormat.parse()方法需要添加异常处理

Test.java

```
public class Test {  
    public static void main(String[] args) throws ParseException {  
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");  
        Date birthday = sdf.parse( "1975-03-03" ); //日期字符串化为Date  
  
        People p=new People( "zz", birthday );  
        p.eat();  
        p.eat("apple");  
  
        System.out.println("年龄： " + p.getAge() );  
  
        SimpleDateFormat sdf2 = new SimpleDateFormat( "yyyy年M月d日" ); //换个格式  
        String birthStr = sdf2.format( p.getBirthday() ); //Date格式化为串  
        System.out.println( "生日： " + birthStr );  
    }  
}
```

```
People can eat  
People can eat apple  
年龄： 46  
生日： 1975年3月3日
```

课后练习题

- 编写一个Point类：拥有x, y坐标值；
- 编写一个Line类：拥有两个点，能计算两点线段长度、斜率；
- 考虑特殊斜率情况：如0或者无穷大
- 编写测试类：增加判断两条线段是否正交功能。

[【返回】](#)

附录1：Date类和Calendar类

- Date类表示的是特定的，瞬间的，它能精确毫秒 (比较晦涩)
- Calendar是一种抽象类，它为Date类与年、月、日等字段之间的转换提供了一些方法
- 在大多数情况下，Date要实现的功能都可以通过Calendar来实现的

Date基本用法:

//创建Date对象, 直接获取本地的当前时间

```
Date date = new Date();
```

```
System.out.println(date); //输出是默认格式
```

```
System.out.println("毫秒:" + date.getTime()); //获得毫秒(自1970年1月1日00:00:00以来)
```

//Date转换为String (格式化串)

//yyyy:四位年, MM: 月份, dd: 日期, HH:24小时制, mm分, ss秒 不足两位的补0

```
SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");
```

```
String dateStr = sdf.format(date);
```

```
System.out.println("Date转String:" + dateStr);
```

SimpleDateFormat两个重要用法

//String转换为Date (记得抛出异常)

```
String dateString = "2021-09-25";
```

```
SimpleDateFormat sdf2 = new SimpleDateFormat("yyyy-MM-dd");
```

```
Date date2 = sdf2.parse(dateString);
```

```
System.out.println("String转Date: " + date2);
```

Tue Aug 31 20:35:04 CST 2021

毫秒:1630413304023

Date转String:2021年08月31日 20:35:04

String转Date: Sat Sep 25 00:00:00 CST 2021

Calendar基本用法:

//getInstance()方法返回一个Calendar对象，其日历字段为当前日期和时间

Calendar cal = Calendar.getInstance(); ← Calendar是抽象类，不能直接new对象

// get()方法获取日历信息

System.out.println("年:" + cal.get(Calendar.YEAR)); // 获得年

System.out.println("月:" + (cal.get(Calendar.MONTH) + 1)); // 月份从0开始，所以取月份要+1

System.out.println("日:" + cal.get(Calendar.DAY_OF_MONTH)); // 获得日期

System.out.println("时:" + cal.get(Calendar.HOUR_OF_DAY)); // 获得时

System.out.println("分:" + cal.get(Calendar.MINUTE)); // 获得分

System.out.println("秒:" + cal.get(Calendar.SECOND)); // 获得秒

//手动设置某个日期

Calendar cal02 = Calendar.getInstance();

//注意，设置时间的时候月份的下标是在0开始的，设置日期3个参数也可以

cal02.set(2020,9,1,12,0,0); //设置为 2020年10月1日 12:00:00

System.out.println("新设置的Date: " + cal02.getTime()); //getTime(): 将Calendar转为Date

```
年:2021
月:8
日:31
时:20
分:43
秒:53
新设置的Date: Thu Oct 01 12:00:00 CST 2020
```

Date和Calendar互相转换：

//Calendar转Date

```
Calendar cal = Calendar.getInstance();
```

```
Date date = cal.getTime();
```

//Date转Calendar

```
Date date2 = new Date();
```

```
Calendar cal2 = Calendar.getInstance();
```

```
cal2.setTime( date );
```

【完】