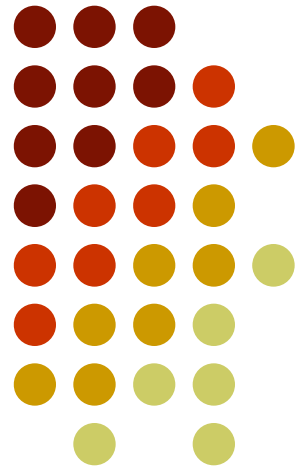




# 第四章 面向对象的 Java程序设计

- 4.1 类和对象
- 4.2 类的方法
- 4.3 类的封装
- 4.4 Java的多态
- 4.5 this和super
- 4.6 构造函数
- 4.7 类的继承
- 4.8 抽象与抽象类
- 4.9 接口的定义与使用
- 4.10 内部类
- 4.11 Java设计模式\*



AI201-CS201-2-20220401  
CS1912-20210407  
AI191-20201014  
CS1812-20201020



# 斯坦福：编程方法学

- <https://open.163.com/newview/movie/free?pid=KFTLSARG6&mid=LFTLSBCL6>
- 「编程方法学」是最大的编程入门课程，也是斯坦福大学主要课程之一。介绍了当代程序设计基本思想：面向对象，模块化，封装，抽象化与测试。  
《编程方法》将良好的编程思想连同广泛应用的Java语言一同讲授。着重教授良好的编程风格和Java语言的特色。
  - ✓ Karel是一门面向初学者的教学编程语言，由Richard E. Pattis在他的书《Karel The Robot: A Gentle Introduction to the Art of Programming》中提出。
  - ✓ Karel机器人

## 4.1 类与对象

- Java编程语言中的抽象数据类型概念被认为是**class**。类给对象的特殊类型提供定义。它规定对象内部的数据，创建该对象的特性，以及对象在其自己的数据上运行的功能。因此类就是一块模板。
- 类是对象的抽象及描述，它是具有统一属性和方法的多个对象的统一描述体，是用来定义一组对象共有属性和方法的模板。
- Java类由**状态**（或属性）和**行为**两部分组成。
- 在Java程序中，用**变量**来描述类的**状态**，用**方法**来实现类的**行为**。方法定义了可以在对象上进行的操作，方法定义类用来干什么。

# 类的示例：学生类

```
1.  public class Student{
2.      // 定义（属性）变量
3.      int id;
4.      String name;
5.      double marks;
6.      boolean pass;
7.      // 定义成员方法
8.      public void display(){
9.          System.out.println("id = "+id);
10.         System.out.println("name = "+name);
11.         System.out.println("marks = "+marks);
12.         System.out.println("pass = "+pass);
13.     }

14.     public static void main(String[] args){
15.         Student s = new Student();
16.         s.display();
17.     }
18. }
```

## Student.class

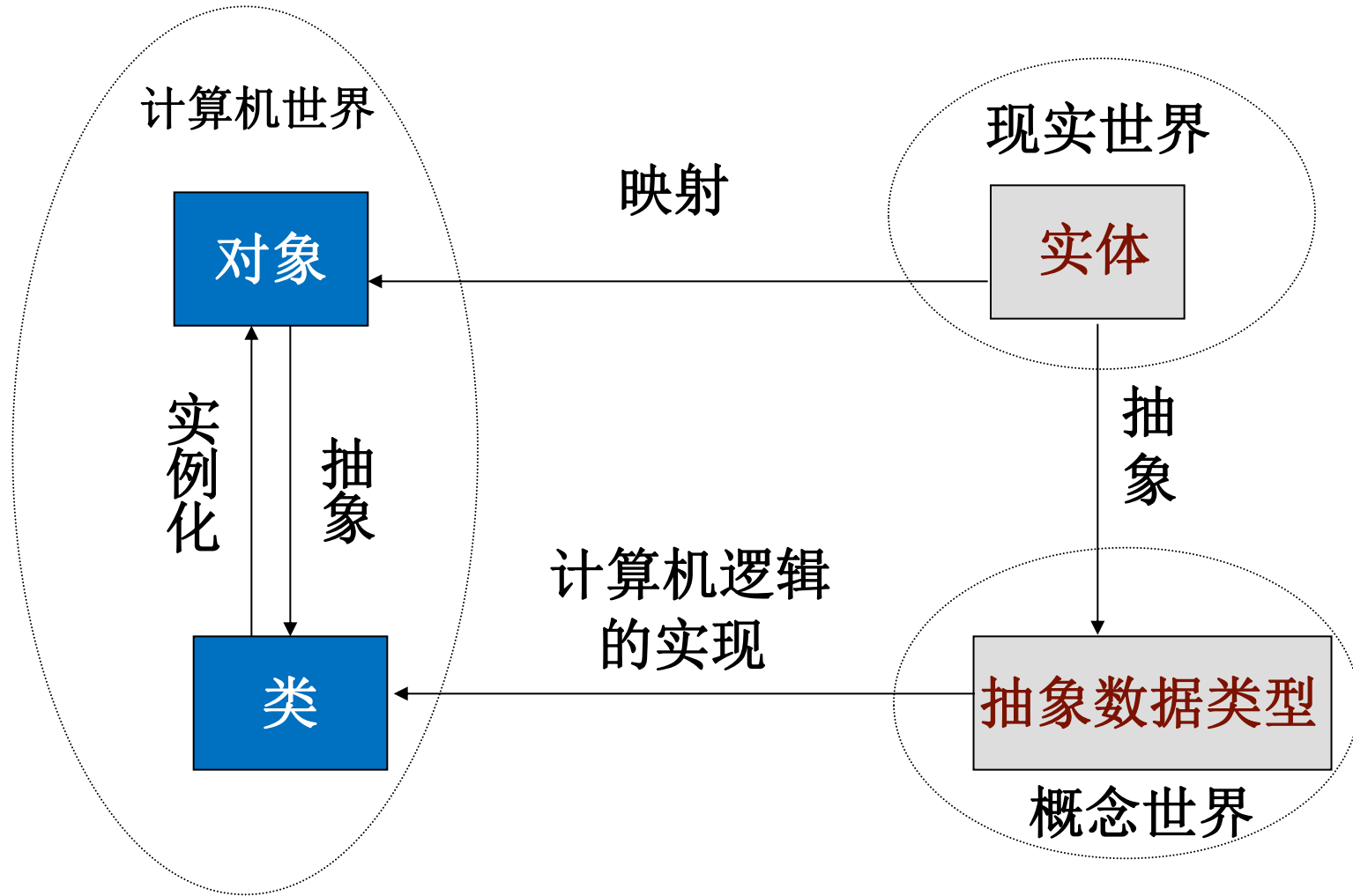
- -id:int
  - -name:String
  - -marks:double
  - -pass:boolean
- +display():void  
+takeTest():void

问题？没有初始数据

# 类的示例：雇员类

```
1. class Employee {  
2.     String name;  
3.     int age;  
4.     float salary;  
5.  
6.     void print() {  
7.         System.out.println(name + "年龄为: " + age + "月薪  
为: " + salary);  
8.     }  
9.  
10.    void upSalary(float inc) {  
11.        salary = salary + inc ;  
12.    }  
13. }
```

# 对象是类的唯一性实例



# Java中的类

## ➤ Java中的类结构

- ✓ 基类：所有Java的类都派生自`java.lang.Object`
- ✓ Java的类可组织在包（**package**）中
- ✓ Java提供的标准类库分布在一系列的包中，如`java.lang`，`java.util`，`java.net`等
- ✓ 编程的过程就是继承基类或标准类而创建、定义特殊子类的过程

<https://docs.oracle.com/en/java/javase/11/docs/api/overview-tree.html>

# 类的说明语法

类的声明格式如下：

[类的修饰符] **class** 类名 [extends 父类名][implement 接口名表]

其中：

- 类的修饰符包括 **public**、**abstract** 和 **final**, 也可以缺省（无修饰符）。

它们的语义及用法见后面介绍；

- **class** 是关键字，用来标识声明类的语句；
- 类名必须是合法的Java标识符，用来标识所声明的类；
- **extends** 是关键字，表示所声明的类继承了指定的父类；
- “**implement 接口名表**” 表示所声明的类继承了指定的接口。



# Java中类的声明

- 类修饰符（Modifier）：
  - ✓ **public**: 允许其他类（没有限制）访问本类，  
一个.java源文件仅可以有一个public类
  - ✓ 缺省：可被当前包中的其他类访问
  - ✓ **abstract**: 没有实例的抽象概念类，是它的所有子类的公共属性和公共方法的集合
  - ✓ **final**: 不能被继承，不能有子类的类；易于安全性的实现，或用于创建固定功能的类
- 父类继承声明: **extends** 父类名
- 实现接口(interface): **implements** 接口名

# Java中类的声明

➤ 例:

```
public abstract class shape{  
    .....  
}  
class rectangle extends shape{ //矩形  
    .....  
}  
final class roundrect extends rectangle{ //椭圆  
    .....  
}
```

注意: **final**可以与**abstract**一起使用吗?

# 举例

```
class Man {  
    String name ;  
    int age;  
    float weight;  
    Man(String name,int age,float weight){  
  
    void walk() { ... }  
    void speak() {... }  
    void sleep() { ... }  
} 0
```

姓名: Jack  
年龄: 28  
体重: 65 kgs

动作:  
走路  
说话  
睡觉

**Man** **Jack**=new **Man**("Jack",28, 65)

# 4.1 类与对象

```
import java.util.Scanner;

public class TemperatureConverter {
    public double getFahrenheit(double celsius) {
        double fahrenheit = 1.8 * celsius + 32; // 计算华氏温度
        return fahrenheit; // 返回华氏温度
    }

    public static void main(String[] args) {
        System.out.println("请输入要转换的温度（单位：摄氏度）");
        Scanner in = new Scanner(System.in); // 获得控制台输入
        double celsius = in.nextDouble(); // 获得用户输入的摄氏温度
        TemperatureConverter converter = new TemperatureConverter(); // 创建类的对象
        double fahrenheit = converter.getFahrenheit(celsius); // 转换温度为华氏度
        System.out.println("转换完成的温度（单位：华氏度）：" + fahrenheit); // 输出转换结果
    }
}
```

方法

对象



# 创建对象

➤ 对象(实例instance): 以类为模板创建的具体实例(实际个体)

✓ 创建某个类的具体对象(类似变量声明)

类名 欲创建的对象名 = **new** 类名(参数表);

例如:

```
String s = new String( "Hello!" );
```



- 实例化一个对象
  - ✓ 现实生活中某个具体的人，即对象“**Jack**”
- 这些变量(**name, age, weight**)被称为类**Man**的成员。
- 其中**Man(String n,int a,float w){ ..... }** 是类的构造函数。



# 类的成员变量的声明

- 声明成员变量(member)的语法  
[修饰符] 数据类型 变量名;
- 引用：对象名.变量名
- 修饰符
  - ✓ 访问控制符
    - **public**: 可被所有其他类引用
    - **private**: 仅可被该类自身引用和修改，不能被其他任何类（包括该类的子类）引用，它提供了最高级别的保护。

# 成员变量的声明

- **protected**: 该类自身、子类、同一包中的其他类
- 缺省: 本包中的类
- ✓ **static**: 类成员变量, 对所有的实例一致, 引用时前缀可使用类名或对象名
- ✓ **final**: 数值不变常量, 定义同时应对其进行初始化

表 4.1 访问权限

访问控制	本类	同一包 中的类	不同包 中子类	任何包中的类
public	√	√	√	√
protected	√	√	√	×
缺省(友元)	√	√	×	×
private	√	×	×	×





# 成员变量的声明

➤ 例：

```
class Shape{  
    private protected int x;  
    private protected int y;  
    static final float MaxArea=100.0f;  
    .....  
}
```

**final 类型 常量名=值**



# 成员变量的声明

```
class Employee {  
    String name ;  
    int age ;  
    float salary ;  
}
```

```
Employee e = new Employee( ) ;  
e.name="张立" ;  
e.age=21;  
e.salary = 528.37 ;
```

实例变量—成员变量  
其生命周期与对象存在的时间相同。

# 类成员变量（静态变量）

- 静态变量——为类的各实例共享的变量。  
**static** 静态变量名；
- 位于类的内存区域中，为该类的各个对象共享。
- 无此限定符的变量是实例变量。

```
class ex {  
    int i ;  
    static int j ;  
    static final int k=10 ;  
    ---  
}
```

- **final** 常量可同时定义为**static**

类ex

j  
k=10

对象1

i : 3

对象2

i : 5



# 类成员变量（静态变量）

- 实现各实例之间的通讯

对于该类的任何一个对象访问该静态变量时，取得的都是相同的值

- 跟踪创建的实例数

```
public class Count {  
    private int serial ;           //实例变量  
    private static int counter = 0 ; //静态变量  
    public Count() {  
        counter++;  
        serial = counter ;  
    }  
}
```

- 静态变量类似于某些语言中的全局变量



# 类成员变量（静态变量）

- 非**private** 的静态变量无须创建实例就可以从类的外部访问。

```
class StaticVar {  
    static int x=100 ;  
}  
public class test {  
    public void m() {  
        int m = StaticVar.x ;  
    }  
}
```



# 类成员变量（静态变量）

```
class PhoneCard200 {  
    static final String connectNumber = "200";  
    static double additoryFee;  
    long cardNumber ;  
    int password;  
    boolean connected;  
    double balance;  
    ...  
}
```



# 类成员变量（静态变量）

```
public class AA {  
    public static void main(String args[]) {  
        PhoneCard200 my200_1 = new PhoneCard200();  
        PhoneCard200 my200_2 = new PhoneCard200();  
        PhoneCard200.additoryFee = 0.1;  
        System.out.println("第一张200卡的接入号码:" +  
                             my200_1.connectNumber);  
        System.out.println("第二张200卡的附加费: " +  
                             my200_2.additoryFee);  
        System.out.println("200卡类的附加费: " +  
                             PhoneCard200.additoryFee);  
        System.out.println("200卡接入号码:" +  
                             PhoneCard200.connectNumber);  
    }  
}
```

## 例3-2: MyDate类的例子

```
//sample MyDate
public class MyDate {
    int day;
    int month;
    int year;
    public static void main(String args[]){
        MyDate today;
        today = new MyDate();
        today.day=25;
        today.month=9;
        today.year=2007;
        System.out.println(today.year+"年"+today.month+"月"+today.day+"日");
    }
    public void tomorrow() {
        // code to increment day
        // like  day++;
    }
}
```





# 对象与类——几种相互关系

## ➤ 包含关系

- ✓ 当对象B是对象A的属性时，我们称对象A包含对象B。

## ➤ 关联关系

- ✓ 当对象A的引用是对象B的属性时，我们称对象A和对象B之间是关联关系。

## ➤ 类之间的继承关系

- ✓ B类继承了A类，就是继承了A类的属性和方法。
- ✓ A类称之为父类，B类称之为子类。

## 4.2 类的方法

方法声明采取这样的格式：

```
<modifiers> <return_type> <name> ([<argument_list>])  
[throws <exception>]  
{  
<block>  
}  
public void addDays(int days) {  
}
```

- 定义对类内成员变量（数据）的操作
- 方法是类的动态属性，标志了类所具有的功能和操作



# 类的方法定义规则

1. 方法名后的小括号是方法的标志。
2. 形式参数是方法从调用它的环境输入的数据。
3. 返回值是方法在操作完成后返还给调用它的环境的数据。
4. 要指定方法返回值类型。如没有返回值，类型要写 **void**。
5. 方法名相同，但参数不同，是不同的方法。
6. 与类名相同的所有方法都是类的**构造函数**(构造方法)。

## 4.2.1 方法的声明

### ➤ 修饰符(Modifier) :

- ✓ 访问控制符：  
public、private、protected、private protected
- ✓ **static**: 静态方法，又称类方法
  - 使用类名或对象名作为前缀，建议使用类名
  - 在方法体中只能使用**static**变量和**static**方法
- ✓ **abstract**: 只有方法头，而没有具体的方法体和操作实现的方法，要在子类中通过重新定义（**override**）来实现
- ✓ **final**: 不能被重新定义（**override**）的方法



# 访问控制符

- 访问控制符是一组起到限定类、域或方法是否可以被程序里的其他部分访问和调用的修饰符。
- 类访问控制符
  - ✓ 公共类 : **public** 类名
  - ✓ 一般类(缺省)
- 一个类被声明为公共类, 表明它可以被所有的其他类所访问和引用。程序的其他部分可以创建这个类的对象、访问这个类的内部可见成员变量和调用它的可见方法
- 一般类只能被同一包中的类访问和引用。
- 定义在同一个程序中的所有类属于一个包。

# 域和方法的访问限定符

- 一个类作为整体对程序的其他部分可见，并不能代表类内的所有成员变量和方法也同时对程序的其他部分可见，前者只是后者的必要条件
- 成员变量和方法的可见性

<b>public</b>	公共变量和公共方法，可被任何类使用。
<b>protected</b>	可被同包中其他类使用。 可被子类继承(包括位于不同包中的子类)
无访问限定符	可被同包中其他类使用。
<b>private</b>	只限在本类内使用。

# 高级访问控制

◆成员变量和方法有4种访问级别：

**public, protected, default(package), private ;**

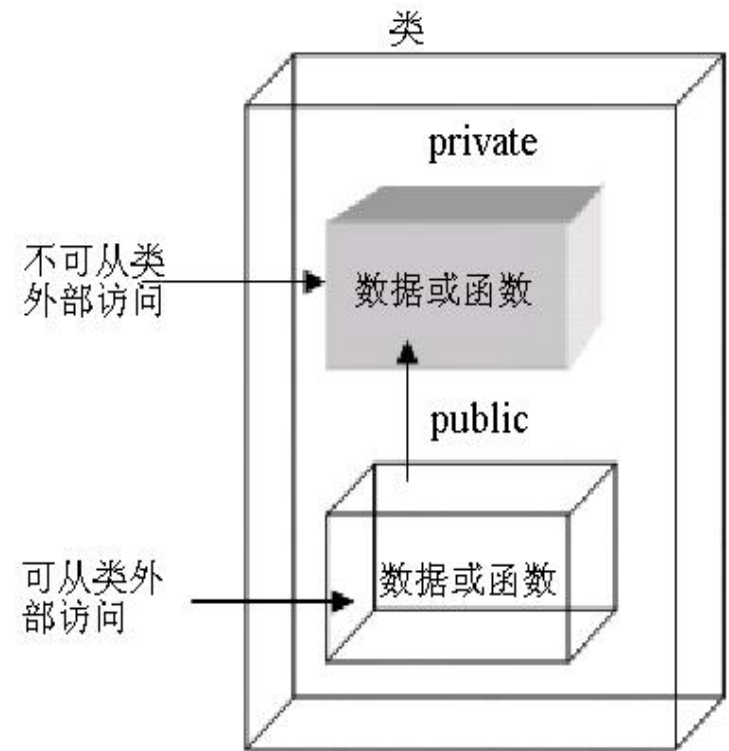
◆类有两种访问级别：**public** 或**default**。

◆修饰符的作用范围：

修饰符	本类中	本包中	包含子类	所有类
<b>Modifier</b>	<b>Same class</b>	<b>Same Package</b>	<b>Subclass</b>	<b>Universe</b>
<b>public</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>
<b>protected</b>	<b>Yes</b>	<b>Yes</b>	<b>Yes</b>	
<b>default</b>	<b>Yes</b>	<b>Yes</b>		
<b>private</b>	<b>Yes</b>			

# 域和方法的访问限定符

- 为了使对象具有良好的封装性，一般将类的实例变量设计成私有。
- 为了使其它类或对象能够访问私有实例变量，本类必须提供访问私有变量的方法（公共方法）。
- 按照惯例：
  - ✓ 读私有变量的方法取名为 `get...`
  - ✓ 写私有变量的方法取名为 `set...`



函数=方法





# 例：学生类及测试类

```
public class Student {  
    private String name; // 定义姓名  
    private String sex; // 定义性别  
    private int age; // 定义年龄  
  
    public Student(String name, String sex, int age) { // 利用构造方法初始化域  
        this.name = name;  
        this.sex = sex;  
        this.age = age;  
    }  
  
    public String getName() { // 获得姓名  
        return name;  
    }  
  
    public String getSex() { // 获得性别  
        return sex;  
    }  
  
    public int getAge() { // 获得年龄  
        return age;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Student std = new Student("李默", "男", 23); // 创建对象  
  
        System.out.println("姓名: " + std.getName()); // 输出姓名  
        System.out.println("性别: " + std.getSex()); // 输出性别  
        System.out.println("年龄: " + std.getAge() + "岁");  
        // 输出年龄  
    }  
}
```



# 方法的声明（续）

- 返回类型：void、return
- 参数列表：参数类型可为类类型
- throws:
  - ✓ 通告本方法中会产生的例外（异常）类型，提醒调用者要对相应的可能异常（例外）进行处理。当执行此方法并发生了异常时，程序会转入调用者编制的异常处理程序段。

# 数据传递

- **<argument\_list>** 允许将参数值传递到方法中。列举的元素由逗号分开，而每一个元素包含一个类型和一个标识符。方法定义中的参数并没有实际值，仅仅是为了描述处理过程而引入的，因此称为**形式参数**（简称**形参**）。使用方法时给出参数的实际值，这些实际值称为**实际参数**（简称**实参**）。

# 方法的参数

- 方法的参数要有名有型，参数的作用域在本方法中，在方法体中可以象方法体自己定义的变量一样使用。
- 参数是值传递  
对象的内容可以改变，但对象的引用决不会改变。
- 方法的参数可以与类的成员变量同名，这时，参数在方法体中将隐藏同名的成员变量。

```
class Circle {  
    int x,y, radius ;  
    setCircle (int x , int y , int radius ){  
        ...  
    }  
}
```

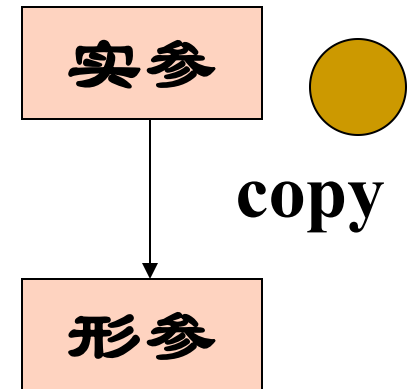


# 参数传递例

```
public class PassTest {  
    float m_float ;  
    void change1(int pi) {  
        pi = 100;  
    }  
    void change2(String ps) {  
        ps=new String("Right") ;  
    }  
    void change3(PassTest po ) {  
        po.m_float=100.0f ;  
    }  
    public static void main(String[] args) {  
        PassTest pt = new PassTest() ;  
    }  
}
```

# 参数传递例

```
int i = 22 ;  
pt.change1( i ) ;  
System.out.println("i value is " + i);  
String s = new String( "Hello" ) ;  
pt.change2( s ) ;  
System.out.println("s value is " + s);  
pt.m_float = 22.0F ;  
pt.change3( pt ) ;  
System.out.println("Current pt.m_float is " + pt.m_float);  
} // main()  
} // class
```



**i value is 22**  
**s value is Hello**  
**Current pt.m\_float is 100.0**



# 方法体的定义

- 在方法中定义的变量称为局部变量。
  - ✓ 其生命周期：执行进入方法体时建立，退出方法时撤消。
  - ✓ 局部变量使用之前必须初始化。
- 在方法体中可以访问所有的类属性，可以调用所有类中定义的方法。
- 方法可以递归调用。



# 方法的说明

- 为Employee类增加打印方法——将数据和代码封装在一个单一的实体中。

```
class Employee {  
    String name;  
    int age;  
    float salary ;  
    void print() {  
        System.out.println(name + “年龄为: ” + age + “月薪  
为: ” + salary);  
    }  
} //class
```



# 对象方法的含义

```
Employee emp = new Employee( );  
emp.name="张立" ;  
emp.age=21;  
emp.salary = 528.37F ;
```

- 一旦对象被建立并被初始化，就可以调用该方法打印这个对象的数据。

```
emp.print();
```

- 对象**emp**执行**print** 方法操作自己的数据。
- 向对象发消息，让对象做某件事。



## 4.2.2 类方法（静态方法）

类方法： **static** 方法名(...){ ...}

使用类方法不用创建类的对象。调用这个方法时，建议使用类名做前缀，而不是使用某一个具体的对象名。

```
public class a {  
    public static void main(String[] args){  
        System.out.println(Math.round(3.54));  
        String s = to_char(2.718);  
        System.out.println("e=" + s );  
    }  
    static String to_char(double x) {  
        return Double.toString(x);  
    }  
}
```



# 类方法（静态方法）

- **main**方法是静态方法，这是为了使系统在没有任何实例化对象之前可以运行一个应用程序。
- 如果**main**方法要调用本类的其它方法：
  1. 将这些方法设计成静态方法
  2. 创建对象，使用对象方法。

否则，报**null**  
或者**static** 不能调用非**static** 方法错误

- 一个静态方法不能被一个非静态方法所覆盖。



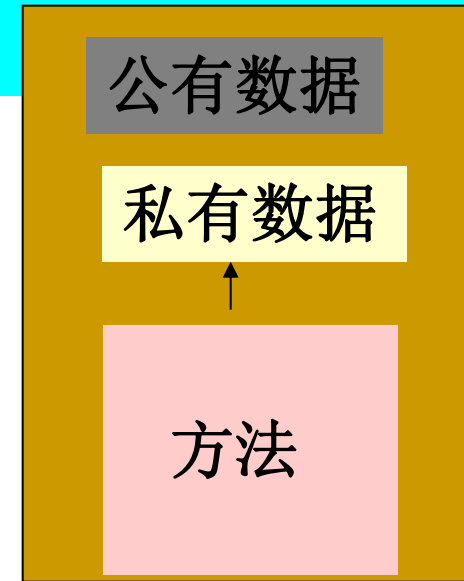
# Bisearch.java

**2021-04-08**

## 4.3 类的封装

封装的实体（对象）

= 数据 + 方法



对象A

- 类的设计者把类设计成一个黑匣子，使用者只能看见类定义的公共方法而看不见方法实现的细节；也不能对类的数据进行操作。

## 4.3 类的封装

- 每个类里都封装了相关的数据和操作。在实际的开发过程中，类多用来构建系统内部的模块。
- 由于封装特性把类内的数据保护得很严密，模块与模块间仅通过严格控制的界面进行交互，使它们之间耦合和交叉大大减少，从而降低了开发过程的复杂性，提高了效率和质量，减少了可能的错误，同时也保证了程序中数据的完整性和安全性。
- 封装使得对象：
  - ✓ 对内成为一个结构完整、可自我管理、自我平衡、高度集中的整体。
  - ✓ 对外则是一个功能明确、接口单一、可在各种合适的环境下都能独立工作的有机的单元。
- 类的封装特性使得类可重用性大为提高，这样的有机单元特别适宜构建大型标准化的应用软件系统，可以大幅度地提高生产效率。



```
class MyDate {
```

```
    private int day;  
    private int month;  
    private int year;
```

```
    public String getDate(){  
        return  
        day+"/"+month+"/"+year;  
    }
```

```
    public int setDate(int a,int b,int c){  
        if ((a>0 &&  
a<=31)&&(b>0&&b<=12)){  
            day=a;  
            month=b;  
            year=c;  
            return 0;  
        }else {  
            return -1;  
        }    }
```

```
public class UseMyDate{
```

```
    public static void main(String[] args){  
        MyDate d = new  
        MyDate();  
        if(d.setDate(22,5,2009)==0){
```

```
            System.out.println(d.getDate());  
        }
```

```
    /*        //illegal operations.
```

```
        d.day=22;  
        d.month = 5 ;  
        d.year = 2009;
```

```
        System.out.println(d.day+"/"+d.mont  
h+"/"+d.year); */  
    }
```



```
1.  class Employee{
2.      private String name;
3.      private int salary;
4.      public Employee(String n, int s){
5.          name = n ;
6.          salary = s;
7.      }
8.      public Employee( String n){
9.          this(n,0);
10.     }
11.     public Employee( ){
12.         this("Unknown");
13.     }
14.     public String getName(){
15.         return name;
16.     }
17.     public int getSalary(){
18.         return salary;
19.     }
20. }
```

```
public class EmployeeTest{
    public static void
    main(String [] args){
        Employee e = new
        Employee();
        System.out.println("Name: "+e.getName()+"
        Salary: "+e.getSalary());
    }
}
```





## 4.3 类的封装

1. 类中的数据使用**private**定义。
2. 使用**public**方法来操作数据。
3. 把逻辑上相关联的数据封装成新的类来使用，类似结构体。
4. 不要定义太庞大的类，要把无关的功能适当分离。
5. 培养良好的编程风格。

## 4.4 Java中的多态

### ➤ 多态 (polymorphism)

用同一个名字调用实现不同操作的方法

方式1: 不同的类之间的同名方法。

如: 中国人/日本人/美国人 说话

方式2: 同一程序中同名的不同方法共存, 参数不同的同名方法

✓ 可以通过子类对父类方法的覆盖实现多态

✓ 可以利用重载(overload)在同一个类中定义多个同名的方法

### ➤ 多态情况下进行方法调用时, 如何区分这些同名的不同方法

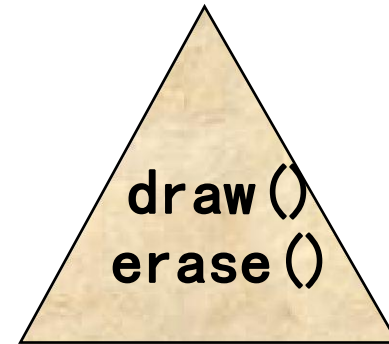
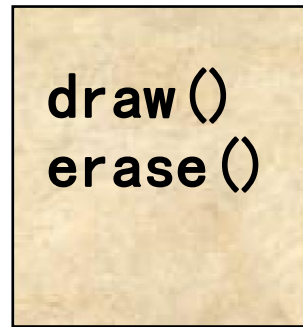
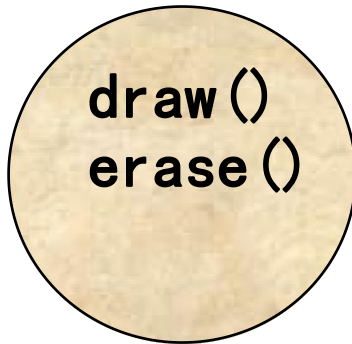
✓ 不同类中的同名方法: 冠以类名以示区别

✓ 同类中的同名方法: 用不同的参数列表(参数的个数, 类型, 顺序的不同)来区别同名的不同方法

# 多态

## ➤ 例：Shapes.java

- ✓ 三个类Circle、Square、Triangle均有draw( )方法和erase( )方法，但不同类的方法的具体操作其实并不相同，实现了多态。



**SL275**

# 方法的重载 (Overloading)

- 例:(相同类型的工作, 不同的参数)

```
public void println(int i);  
public void println(float f);  
public void println(String s);
```

- Integer类

```
String toString();  
static String toString(int i);
```

- Double 类

```
String toString();  
static String toString(double d);
```

在一个类中, 不可以存在两个只有返回值不同的方法

# 例子

//例如：在MyDate.java中设置日期setDate的方法。

```
public void setDate(int yy, int mm, int dd){
    year=yy;
    month=mm;
    day=dd;
}
public void setDate(int yy, int mm){
    year=yy;
    month=mm;
}
public void setDate(int dd){
    day=dd;
}
public void setDate(String cMM){
    if (cMM.length()>=1 && cMM.length()<=2){
        month=Integer.parseInt(cMM);
    }
}
```



```
if (args.length==1) {  
    today.setDate(args[0]);  
}  
if (args.length==2) {  
  
    today.setDate(Integer.parseInt(args[0]),Integer.parseInt(args  
[1]));  
}  
if (args.length==3) {  
  
    today.setDate(Integer.parseInt(args[0]),Integer.parseInt(args[1]),  
Integer.parseInt(args[2]));  
}  
  
today.showDate();
```

## 4.5 this引用和super引用

- **this**和**super**是常用来指代子类对象和父类对象的关键字。Java系统默认，每个类缺省地具有**null**、**this**和**super**三个域，所以在任意类中都可以不加说明而直接使用它们。
- **this**表示的是当前对象本身，更确切地说，**this**代表了当前对象的一个引用。利用**this**可以使用当前对象的域，调用当前对象的方法。
- **super**表示的是当前对象的直接父类对象，是当前对象的直接父类对象的引用。



# this引用

this 指代对象可以用于解决实例变量被局部变量屏蔽的问题

```
public class MyDate {    //类的封装
    private int year, month, day;
    public MyDate (int year,int month,int day){
        this.year = year;
        this.month = (((month>=1)&& (month<=12)) ? month : 0);
        this.day = (((day>=1) &&(day<=31))?day:0);}
    public setDay (int day) {
        this.day = (((day>=1) && (day<=31)) ? day: 1); }
}
```

**date1=new MyDate(2002,5,8)** 构造函数中的 **this.year** 就是指指的是 **date1**的实例变量**year**

2020-10-20 cs 二



# Super引用

```
class Point{
    int x, y;
    Point(int x, int y){
        this.x=x;    this.y=y;
        System.out.println("父类构造函数被调用！");
    } }
class Circle extends Point {
    int r;
    Circle(int r, int x, int y){
        super();
        //    super(x, y);    //先调用父类的构造函数
        this.r=r;
        System.out.println("子类构造函数被调用！");
    }
}
```

通过**super**  
调用超类  
的构造函数

原因：

子类不能继  
承超类的构  
造函数。

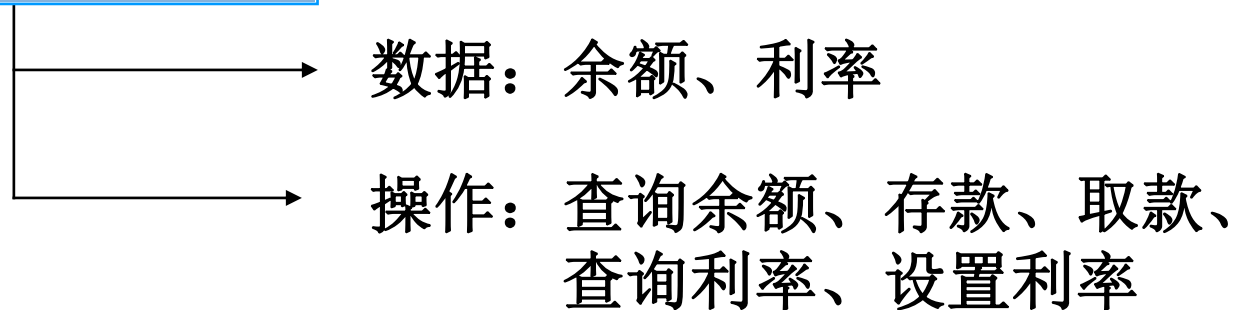
# 例子：银行业务

- 要求：处理简单帐户存取款、查询
- 面向对象的**分析**（OOA）
  - ✓ 对象模型抽象：银行帐户
  - ✓ 对象模型分析：余额、存、取、查
- 面向对象的**设计**（OOD）
  - ✓ 对象模型的实现：类及其成员定义
    - **BankAccount**类
    - **Banlance, etc**
    - **getBalance(), getMoney(), etc**

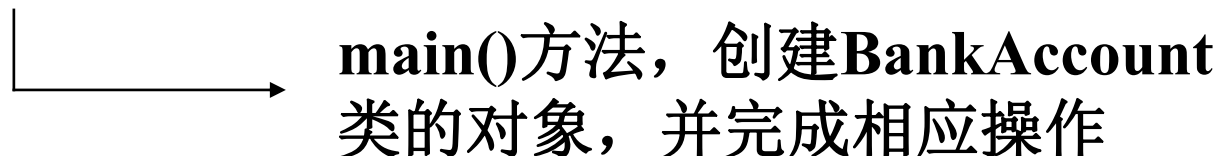
# 银行业务的程序结构

银行帐户类BankAccount

**UseAccount.java**



主类UseAccount





# Account类（上机练习）

Account
-id: int -balance: double -annualRate: double -dateCreated: Localdate
+Account() +Account(id: double, balance: double) +getId(): int +setId(int id): void +getBalance(): double +setBalance(double balance): void +getAnnualRate(): double +setAnnualRate(double annualRate):void +getDateCreated(): LocalDate +getMonthlyInterestRate(): double +withdraw(double amount): void +deposit(double amount): void

账户的id  
账户的余额  
存款的年利率  
账户创建日期  
默认构造方法  
带参数构造方法  
返回id的方法  
修改id的方法  
返回 balance的方法  
修改 balance的方法  
返回 annualRate的方法  
修改 annualRate的方法  
  
返回账户创建日期的方法  
返回月利率的方法  
取款的方法  
存款的方法

作业将在多线程（  
CH07）布置

2020-10-14 AI

2021-04-15 CS1912

## 4.6 构造函数 (constructor)

### ➤ 问题的提出


- ✓ **Java**试图在每个对象一诞生时，就给它一确定的状态，但有时这种系统的缺省初值不合适；所以在创建一个新对象时，需用构造函数完成合法的初始化，使类内成员的初始化工作不依赖于外界完成。

**构造函数是在创建对象的时候自动调用的一个方法，该方法与类名同名。**

**作用：初始化对象的数据属性。**

# 构造函数

```
public class MyDate {  
    int day;  
    int month;  
    int year;  
    public MyDate() {  
        day=6;  
        month=10;  
        year=2008;  
    }  
    .....  
}
```



// 构造函数的方法名与类名相同

# 构造函数

- 构造函数是一种特殊的成员方法，在创建每个新的类对象时自动执行，以保证新对象的各成员有合法、确定的数值。
- 构造函数的名字就是类的名字。
- 构造函数没有返回类型。
- 主要用于完成变量的初始化；另外，一个类中可以存在多个构造函数(重载)，这些构造函数之间可以互相调用，当一个构造函数调用另一个构造函数时，要使用关键字 **this**，同时这个调用语句应该是整个构造函数的第一条可执行语句。



# 例子：Student类和构造函数

```
public class Student {  
    private String name; // 定义姓名  
    private String sex; // 定义性别  
    private int age; // 定义年龄  
    public Student(String name, String sex, int age) { // 利用构造方法初始化域  
        this.name = name;  
        this.sex = sex;  
        this.age = age;  
    }  
  
    public String getName() { // 获得姓名  
        return name;  
    }  
    public String getSex() { // 获得性别  
        return sex;  
    }  
    public int getAge() { // 获得年龄  
        return age;  
    }  
}
```





# 构造函数

- 构造函数只能在创建类对象时 **由系统隐含调用**，而不能直接引用；
- 类对象创建时，利用运算符 **new** 在内存中开辟专用空间，存放指定的类的实例（即对象），这时会自动执行类的构造函数，初始化新对象的成员变量。

# 构造函数

- 实例化对象——初始化对象  
类名 对象名=new 构造函数();  
`Employee emp = new Employee(...);`
- `new`运算符的参数就是调用类的构造函数，初始化新建的对象。
- `new`运算符为对象分配内存空间，并返回对该对象的一个引用。
- 当没有定义任何构造函数时，系统分配一个隐含的构造函数(如：`Employee()`) 里面没有内容。
- 一旦定义了一个构造函数，隐含构造函数失效！



# 定义构造函数

```
Employee(String n,int a,float s){  
    name = n ;  
    if(a>=18 && a<=60)  
        age = a ;  
    else {  
        System.out.println("illegal age");  
        System.exit(1);  
    }  
    salary = s;  
}
```



# 例: SimpleConstructor.java

```
class Rock {  
    Rock(int i) { // constructor  
        System.out.println("Creating Rock number " + i);  
    }  
}
```

```
public class SimpleConstructor {  
    public static void main(String[] args) {  
        for(int i = 0; i < 10; i++)  
            new Rock(i); //create instance  
    }  
}
```

Creating Rock number 0  
Creating Rock number 1  
Creating Rock number 2  
Creating Rock number 3  
Creating Rock number 4  
Creating Rock number 5  
Creating Rock number 6  
Creating Rock number 7  
Creating Rock number 8  
Creating Rock number 9

# 对象初始化三步曲

1. 当为对象分配内存空间后，首先将属性单元置为：

- ✓ 0 （数值型，字符型）
- ✓ false (逻辑型)
- ✓ null (引用型)

对象建立后，有明确的状态，属性都有值。

2. 执行显式初始化

```
class A {  
    private int x=10;  
    private String s = new String("first");  
    private MyTime mt = new MyTime();  
}
```

3. 执行构造函数

# 构造函数之间的调用

```
public class Employee {  
    Employee(String n,int a,float s){  
        name = n ;  
        if(a>=18 && a<=60)  
            age = a ;  
        else {  
            System.out.println("illegal age");  
            System.exit(1);  
        }  
        salary = s ;  
    }  
    Employee( String n , int a ) {  
        this(n , a , 300 ) ;  
    }  
}
```

调用：构造函数  
之间可以互相调用



# 构造函数之间的调用

- 在一个构造函数中，当使用`this` 并给它一个参数列表时，`this`显式调用参数匹配的另一个构造函数。
- 在一个构造函数中，不能象调用一般方法那样调用另一个构造函数。构造函数不是用来调用的，而是`new`算符的参数。

## 4.7 继承

- 继承是一种由已有的类创建新类的机制，是面向对象程序设计的基石之一。
- 子类继承父类的非私有属性和方法，还可以定义新的属性和方法，改写父类的方法。
- 继承实际上是存在于面向对象程序中的两个类之间的一种关系。
- **Java**要求每个类都有父类(隐含为java.lang包中的Object类)。
- **super** 是对父类的引用，引用父类的构造函数、父类成员属性和方法。





## 4.7 Java中类的继承

- 创建类时指明它为某存在类的子类
  - ✓ **extends** 超类/父类名
- 父类
  - ✓ 来自系统类库
  - ✓ 用户自定义类
- 子类将继承除 **private** 外所有父类成员
- **Java不支持多重继承**（但可用接口实现）

# Java中类的继承

- 子类对父类可做扩展和特殊化
  - ✓ **创建新的成员**：变量和方法
    - m\_INextSaveAccNum, m\_INextCheckAccNum
  - ✓ **重新定义父类中已有的变量**：**隐藏**
    - m\_dInterestRate
  - ✓ **重新定义父类中已有的方法**：**覆盖**(override )  
子类中的方法应与父类中的被覆盖的方法有完全相同的：参数列表、返回值
    - setInterestRate( )

# 类的继承

## ➤ 引用类的成员变量或方法时的前缀

- ✓ **this**: 当前类的方法或变量
- ✓ **super**: 直接父类的成员（只能上溯一层）

例: **BankAccount**类      ➔ **SaveAccount**子类

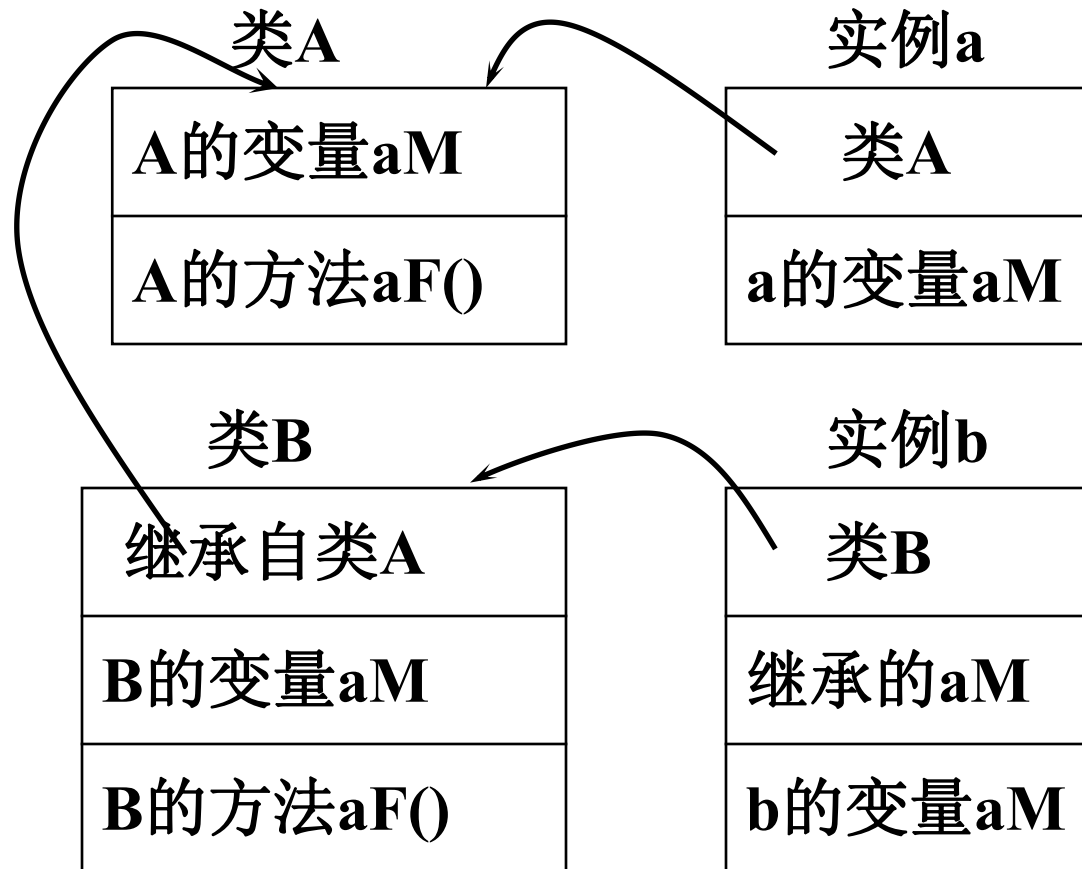
**this.Fee()**: 调用子类的收取费用方法

**super.Fee()**: 调用父类的收取费用方法

## ➤ 子类对象可类型转换成父类对象（upcast）

**BankAccount MyBa = (BankAccount)MySa;**

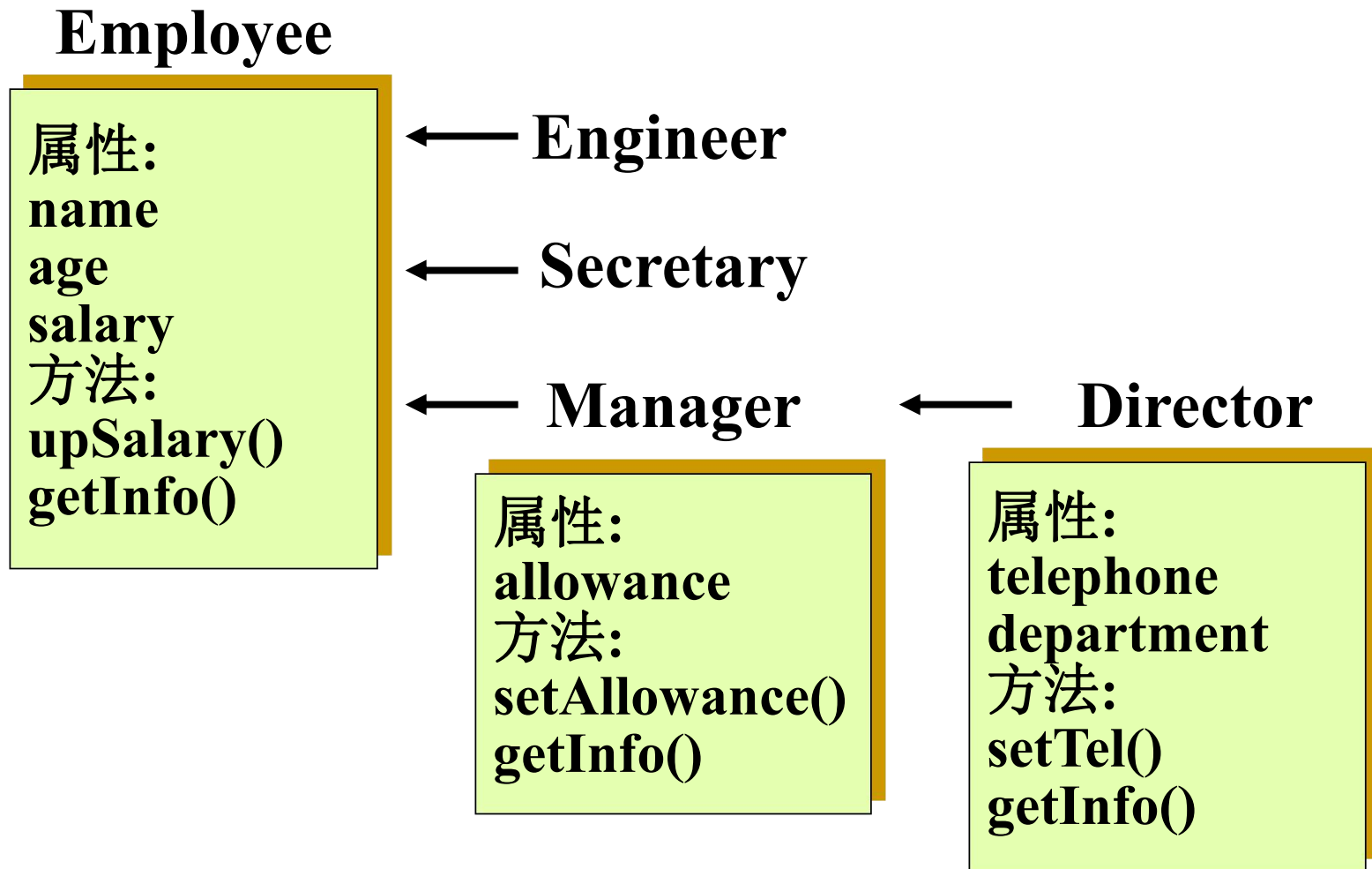
# 类的继承



```
A a=new A( );
B b=new B( );
```

```
a.aM    a.aF()
b.aM    b.aF()
```

# 继承





# 增加属性和方法

```
class Employee {  
    String name ; int age ; float salary ;  
    void upSalary(float inc) {  
        salary = salary + inc ;  
    }  
}  
class Manager extends Employee {  
    float allowance ;  
    void setAllowance(float a) {  
        allowance = a ;  
    }  
}  
class Director extends Manager {  
    String telephone ,department ;  
    void setTel(String tel) {  
        telephone = tel ;  
    }  
}
```



# 构造函数的继承

- 子类或定义自己的构造函数，或使用隐含构造函数。
- 子类没有定义构造函数时，自动继承父类不含参数的构造函数，并在创建新子类对象时自动执行。
- 子类有自己的构造函数时，创建新子类对象时也要先执行父类不含参数的构造函数，再执行自己的构造函数。

# 例： Sandwich.java

```
class Meal { Meal(){... ...} }
class Bread { Bread(){... ...} }
class Cheese { Cheese(){... ...} } //奶酪
class Lettuce { Lettuce(){... ...} } //莴苣, 生菜
class Lunch extends Meal { Lunch(){... ...} }
class PortableLunch extends Lunch { //快餐、便当盒饭 (台湾)
    PortableLunch(){... ...} }
class Sandwich extends PortableLunch {
    Bread b = new Bread();
    Cheese c = new Cheese();
    Lettuce l = new Lettuce();
    Sandwich() {... ...}
    public static void main(String[] args) {
        new Sandwich();
    }
}
```





# 构造函数的继承

- 注意构造函数的调用顺序：
  - ✓ 按继承顺序依次调用父类的不含参数的构造函数，直到到达本子类
  - ✓ 依次执行本子类成员对象的构造函数
  - ✓ 最后，执行本子类的构造函数

**Meal()** —> **Lunch()** —> **PortableLunch()** —> **Bread()**  
—> **Cheese()** —> **Lettuce()** —> **Sandwich()**



# 构造函数的继承

- 子类的构造函数定义中，如要调用父类的含参数的构造函数，需用**super**关键字，且该调用语句必须是子类构造函数的第一个可执行语句。
- 若子类的构造函数中没有**super(...)**语句，系统将隐含调用父类无参数构造函数。
- 一构造函数可利用**this**调用本类其他的构造函数，此时**this**应是第一个可执行语句



# 调用父类构造函数

## ➤ 在继承情况下的对象初始化：

- ✓ 为对象分配内存并初始化(0、 **false**、 **null**)。

如果没有为类中的某些成员赋初始值，**Java**系统会为类的成员赋固定的初始值（数值变量的值为0，布尔变量的值为**false**，未初始化的引用为**null**）。

- ✓ 执行每一层的类的显式初始化。
- ✓ 执行每一层的类的构造函数（由于显式或隐式调用）。

**2022-04-01 #5**

## 4.8 抽象与抽象类

- 程序员可以先创建一个定义共有属性和方法的一般类，再从一般类派生出具有特性的新类。
- 抽象类实际上是所有子类的公共域和公共方法的集合，而每一个子类则是父类的特殊化，是对公共域和方法在功能、内涵方面的扩展和延伸。
- 所有的对象都是通过类来描绘的，但是反过来却不是这样。
- 例如：图形编辑时就会发现问题领域存在着圆、三角形这样一些具体概念，它们是不同的，但是它们又都属于形状这样一个概念，形状这个概念在问题领域是不存在的，它就是一个抽象概念。正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能够实例化的。

# 抽象与具体

## ➤ 抽象

- ✓ 形状
- ✓ 傻瓜
- ✓ 老师
- ✓ 学生
- ✓ .....

## ➤ 对象

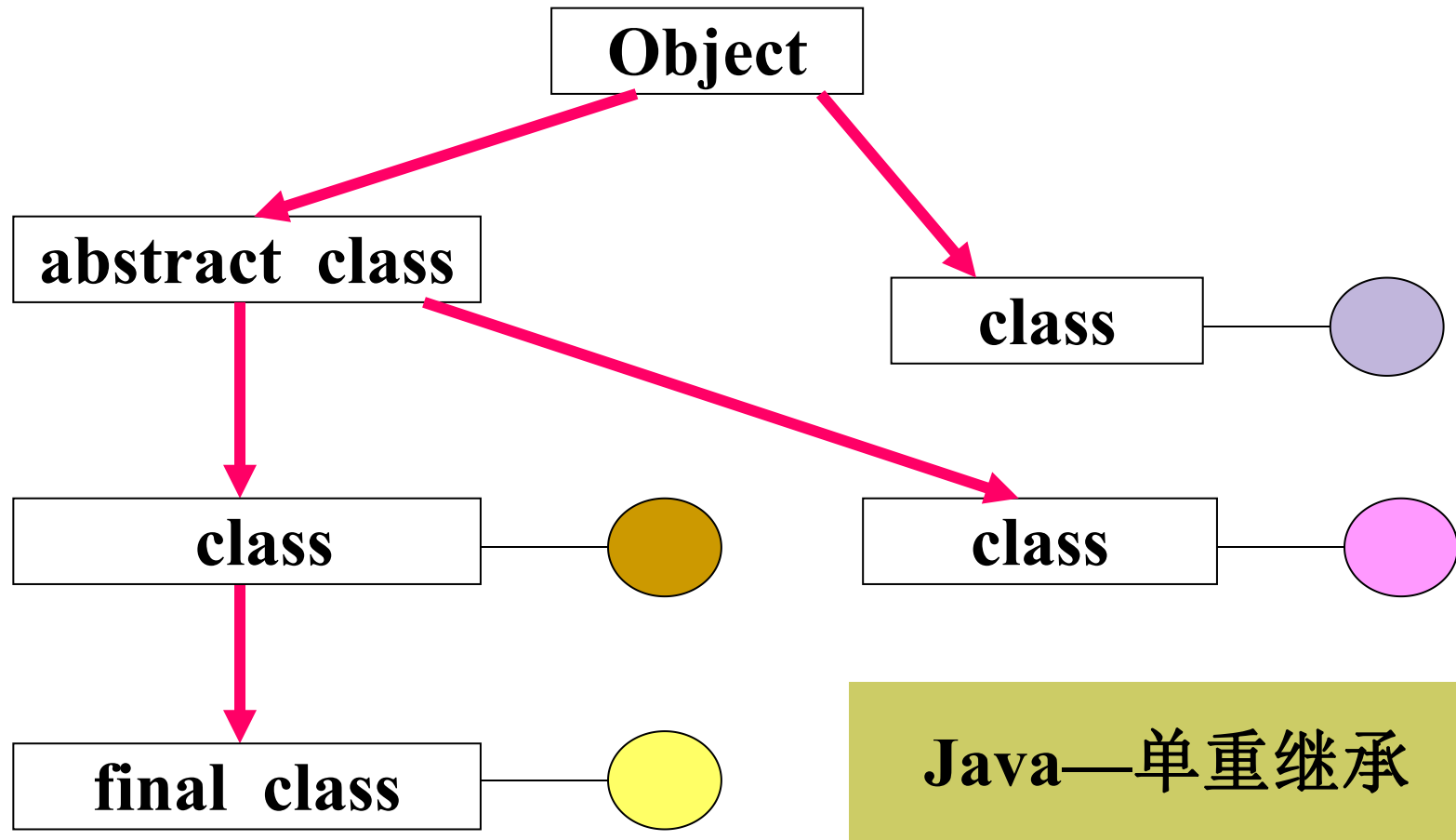
- ✓ 长方形、圆、球
- ✓ 阿Q
- ✓ 孔子
- ✓ 张生
- ✓ .....



# 抽象类

- 用**abstract** 关键字来修饰一个类时，这个类叫作抽象类。抽象类是它的所有子类的公共属性的集合，是包含一个或多个抽象方法的类。抽象类可以看作是对类的进一步抽象。在面向对象领域，抽象类主要用来进行类型隐藏。

# 类继承树



# //1. 抽象类运用的一个例子

```
abstract class Shape{                                //抽象父类
    abstract double area();
}
class Circle extends Shape{
    public int r;
    Circle(int r) { this.r=r;}
    public double area() { return 3.14*r*r; }
}
class Rectangle extends Shape{
    public int width,height;
    Rectangle (int w, int h) { width=w;height=h; }
    public double area() { return width*height; }
}
```





## // 2. 抽象类的运用

```
class ShapeDemo{
    public static void main(String arg[]){
        double shape_area=0;
        Shape c1=new Circle(2);
        Shape c2=new Circle(3);
        Shape r1=new Rectangle(2,3);
        Shape s1[ ]=new Shape[3];
        s1[0]=c1;
        s1[1]=c2;
        s1[2]=r1;
        for (int i=0;i<3;i++)
        { shape_area+=s1[i].area();
        }//求圆和矩形的面积总和。
        System.out.println(shape_area);
    }
}
```



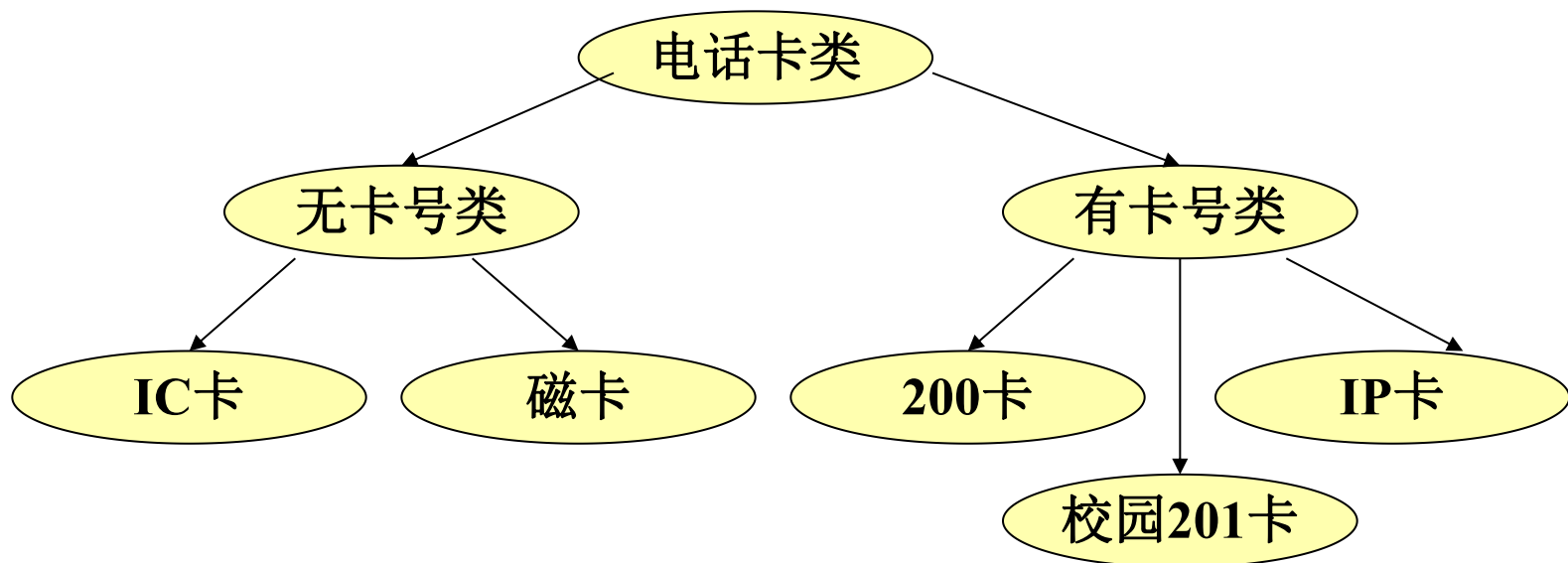
# java抽象类的作用

- 1、抽象类通常都是用来表征对问题领域进行分析、设计中得出的抽象概念，是对大多数看上去不同，可是本质上却是相同的具体概念的抽象。
- 2、在面向对象方法当中，抽象类主要是用来进行类型隐藏。
  - 构造出一个固定的一组行为的抽象描述，可是这组行为却可以有任意个可能的具体实现方式，这个抽象描述就是我们所说的抽象类。这一组任意个可能的具体实现就表现为所有可能的派生类
  - 模块能够操作一个抽象体
  - 因为模块依赖于一个固定的抽象体，所以它可以是不允许修改的
- 3、通过从这个抽象体派生，也能够扩展这个模块的行为功能。
  - 为了可以实现面向对象设计的一个最核心的原则，也就是OCP，抽象类可以说是这当中一个很关键的东西。

## 模板作用

# 抽象类与最终类

- **abstract** 抽象类 — 没有具体对象的概念类
  - ✓ 较高层次的概括
  - ✓ 抽象类的作用是让其他类来继承它的抽象化的特征。
  - ✓ 在程序中不能用抽象类作为模板来创建对象。
- **final** 最终类 — 该类不能有子类。



# 抽象类与最终类

```
abstract class PhoneCard {  
    .....  
}  
class Number_PhoneCard extends PhoneCard {  
    .....  
}  
final class D200_Card extends  
Number_PhoneCard{  
    .....  
}
```

- 抽象类经常要包含某些抽象方法（方法的具体实现要在子类中实现）。
- **final**可以与**abstract**一起使用吗？

不能同时修饰同一个方法，因为**abstract**和**final**修饰词是相对矛盾的，**abstract**必须有子类来实现，而**final**则不能被子类所重载。

# 抽象方法

## ➤ abstract 抽象方法

仅有方法头而没有方法体的方法，为该类的子类定义一个方法的接口标准。

**abstract** void performDial ();

	抽象类	一般类
抽象方法	✓	X
非抽象方法	✓	✓

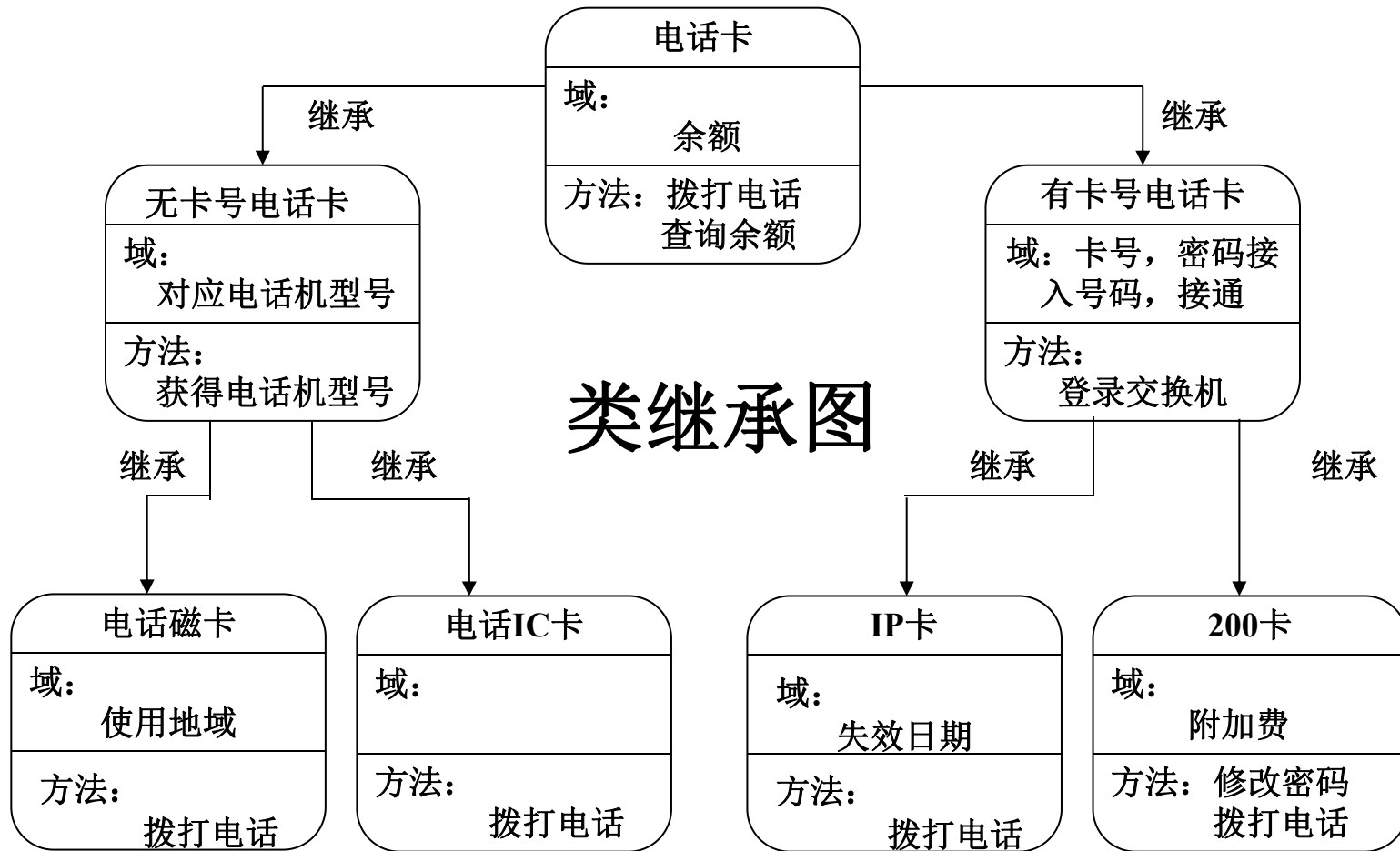
一个抽象类的子类如果不是抽象类，则它必须为父类中的所有抽象方法书写方法体。









# 最终方法

- **final 最终方法**
  - ✓ 不能被当前类的子类重新定义的方法。
  - ✓ 子类不能再重新定义与此方法同名的自己的方法，而仅能使用从父类继承来的方法。只能继承但不能修改。
- 固定了这个方法所对应的具体操作，可以防止子类对父类关键方法的错误的重定义，保证了程序的安全性和正确性。

# 变量的继承与扩充



# 变量的继承与扩充

<b>PhoneCard类</b>	<b>double balance</b>	
<b>Number_ PhoneCard类</b>	<b>double balance; long cardNumber; int password; String connectNumber; boolean connect;</b>	    
<b>D200_Card类</b>	<b>double balance; long cardNumber; int password; String connectNumber; boolean connect; double additoryFee;</b>	



# 变量的隐藏

```
class D200_Card extends Number_PhoneCard {  
    double additoryFee;           //新增  
    double balance;               //隐藏  
    boolean performDial() {  
        if( balance > (0.5 + additoryFee) ) {  
            balance -= (0.5 + additoryFee);  
            return true;  
        }  
        else  
            return false;  
    }  
}
```

**balance**

**balance**

定义自己的变量  
隐藏了父类变量

# 变量的隐藏

- 如何访问被隐藏的父类的变量：
  - ✓ 调用从父类继承的方法操作的是从父类继承的变量。
  - ✓ 使用`super.变量名`。

```
class ExtendAB {  
    public static void main(String[] args){  
        a a1 = new a();    a1.print();        // 1  
        b b1 = new b();    b1.print();        // 1 1 100  
    }  
}
```

```
class a {  
    int x = 1;  
    void print(){  
        System.out.println(x);  
    }  
}
```

```
class b extends a{  
    int x=100;  
    void print(){  
        System.out.println("super.x= "  
                             + super.x);  
  
        super.print();  
        System.out.println("x= " + x);  
    }  
}
```

# 方法覆盖(overriding)

- 在面向对象的程序设计中，子类可以把从父类那里继承来的某个方法改写，形成同父类方法同名、解决的问题也相似、但具体实现和功能却不尽一致的新方法。
- 定义与父类完全相同的方法，实现对父类方法的覆盖：
  - ✓ 完全相同的方法名
  - ✓ 完全相同的参数列表
  - ✓ 完全相同类型的返回值
  - ✓ 注意：在满足上述三个条件同时，还必须保证：  
访问权限不能缩小，抛出的例外要相同
- 三个条件有一个不满足，就不是方法的覆盖，而是子类自己定义与父类无关的方法，父类的方法未被覆盖，因而仍然存在。
- 调用父类被覆盖的方法— `super.方法名`

# 方法覆盖—抽象方法的实现

- PhoneCard类抽象了一个其子类共有的方法：  
`abstract boolean performDial();`
- 这个方法代表的相同功能（拨打电话），但在不同种类的电话卡中，其具体实现是不同的。不同的子类可以重新定义该方法。
- 但是所有的类中，凡是实现拨打电话这种方法的功能的方法，尽管内容不同，却共享相同的名字——`performDial`
- 由于同名的不同方法是属于不同的类，所以只需在调用方法时指明是哪个类的方法即可区分（类名.方法名或对象名.方法名）。



# 方法的覆盖

abstract PhoneCard类:

abstract boolean performDial();

double getBalance()

abstract Number\_PhoneCard类:

abstract boolean performDial();

double getBalance()

boolean performConnection(long cn,int pw)

final D200\_Card类:

- boolean performDial() 覆盖 实现抽象方法
- double getBalance() 覆盖 实现细节不同
- boolean performConnection(long cn,int pw)



# 方法重载/方法覆盖/变量隐藏

- 方法的覆盖与方法重载的区别
  - ✓ 重载：一个类中，同名方法（参数不同）
  - ✓ 覆盖：子类对父类方法的覆盖（不同类的同名方法）
- 方法的覆盖与变量的隐藏的区别：
  - ✓ 子类隐藏父类的变量只是使之不可见，父类的同名变量在子类对象中仍然占有自己独立的内存空间；
  - ✓ 而子类方法对父类同名方法的覆盖将清除父类方法占用的内存，从而使父类方法在子类对象中不复存在。



# 类的初始化-静态初始化器

- 由关键字**static**引导的一对大括号括起的语句组。用来完成类的初始化的工作，作用与构造函数相似
- 与构造函数的区别：
  - ✓ 构造函数是对每个新创建的对象初始化，而静态初始化器是对类自身进行初始化；
  - ✓ 构造函数是在用**new**运算符产生新对象时由系统自动执行，而静态初始化器则是在它所属的类加载入内存时由系统调用执行；
  - ✓ 不同于构造函数，静态初始化器不是方法，没有方法名、返回值和参数列表。

# 静态初始化器

- 静态数据成员、复杂数据成员的初始化可在静态初始化器中完成
- 静态初始化器中的语句在类加载内存时执行，这种初始化操作要比构造函数执行得更早。
- ```
static {  
    a=10;  
    b=a+3;  
    c=a+b;  
    .....  
}
```





# 垃圾回收机制

- 一个小问题？

```
Rock R1=new Rock(1);
```

```
Rock R2=R1;
```

- 这时，内存中为**Rock**类创建了几个实例呢？
- 如果只有一个实例，那么这个实例有几个引用（reference）呢？

# 垃圾回收机制

- **Java**的垃圾回收机制：每个对象有自己的**REFERENCE**计数器,计数器减为零时对象丢失，系统在其后某个时刻自动回收不再可用的对象。

```
{Rock R1=new Rock(1); //计数为1
```

```
    {Rock R2=R1; //计数为2
```

```
        ... ..
```

```
    } //计数为1
```

```
        ... ..
```

```
    } //计数为0
```



## 4.9 接口的定义与使用

接口是和类很相似而又有区别的一种结构，接口的设计和调用也是Java程序设计的重要技术。从结构上，接口有如下特点。

- (1) 接口用关键字 **interface** 来定义，而不是用 **class**。
- (2) 接口中定义的变量全是最终的静态变量。
- (3) 接口中没有自身的构造方法，而且定义的其他方法全是抽象方法，即只提供方法的定义，而没有提供方法的实现语句。
- (4) 接口采用多重继承机制，而不是采用类的单一继承机制。



# 例： ActionListener接口

```
import java.awt.*;
import java.awt.event.*;
public class Calcu implements
ActionListener {
    private Frame f;
    private Panel p;
    private Button b1,b2,b3,b4;
    private TextField t1;

    public static void main(String
        args[ ]){
        Calcu caculator = new Calcu();
        caculator.go();
    }
```

```
public void go() {
    f= new Frame("Caculator");
    f.setSize(500,300);
    p = new Panel();
    p.setLayout(new GridLayout(2,2));
    p.setBackground(Color.red);
    f.add(p,"South");
    t1= new TextField(" ",20);
    f.add(t1,"North");
    b1=new Button("1");
    b2=new Button("2");
    b3=new Button("3");
    b4=new Button("4");
    p.add(b1);
    p.add(b2);
    p.add(b3);
    p.add(b4);
    f.pack();
    f.setVisible(true);
}
```



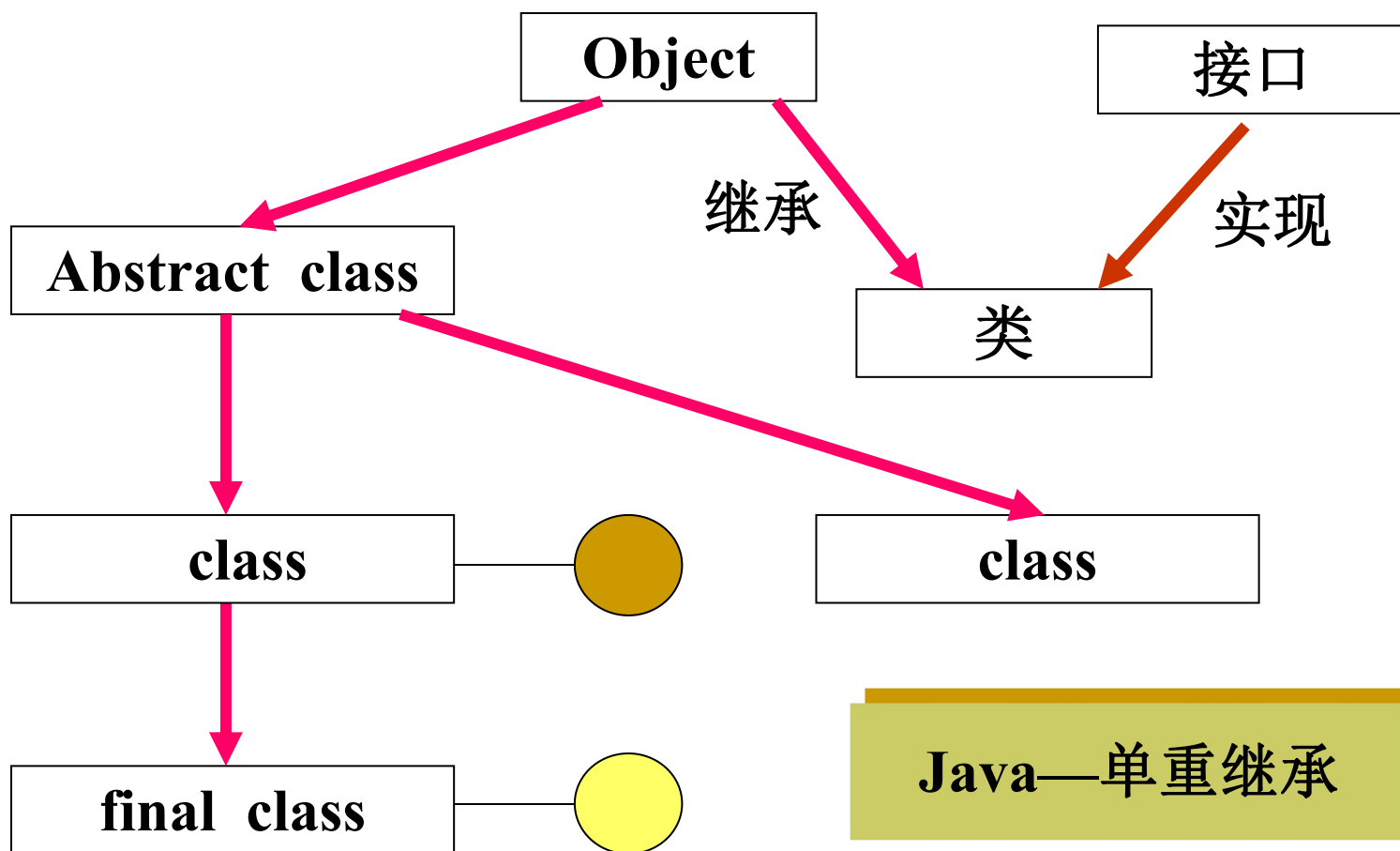
## 4.9 接口的定义与使用

```
b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);
}
public void actionPerformed(ActionEvent e){
    t1.setText(e.getActionCommand());
    System.out.println("Action now"+
e.getActionCommand() );
} }
```

## 4.9 接口的定义与使用

- “接口”是抽象类的概念。
- 接口中的方法都是没有方法体的抽象方法。
- 接口中只能定义 **static final** 属性。
- 接口定义的仅仅是实现某一特定功能的一组方法的对外接口和规范，而并没有真正地实现这个功能。
- 接口的功能实现是在“继承”了这个接口的各个类中完成的，由这些类来具体定义接口中所有抽象方法的方法体。
- 通常把对接口的“继承”称为“实现”。

# 接口的实现



## 4.9 接口（interface）

- 接口是一种特殊的类，是作为一个整体声明的抽象方法和常量
  - ✓ 方法：抽象、无定义的（不必写abstract）
  - ✓ 常量：隐含为final和static的
- 接口定义

```
public interface 接口名 [extends 父接口名列表]
{ // 常量域声明
    public static final 域类型 域名 = 常量值;
    // 抽象方法声明
    public abstract 返回值类型 方法名( 参数列表 );
}
```





# 接口（interface）

- 使用接口
  - ✓ 当一个类声明 **implements** 某一 **interface** 时，必须具体实现该 **interface** 中定义的抽象方法，并把此方法定义为**public**的。
- 利用接口可实现多重继承
- 便于设计更合理的类层次，代码更灵活

# 接口的实现

```
public class MyApplet extends Applet
    implements Runnable , MouseListener {
    .....
}
```

- 一个类只能有一个父类，但是它可以同时实现若干个接口。如果把接口理解成特殊的类，那么这个类利用接口实际上就获得了多个父类，即实现了多继承。
- **instanceof** 运算符可用来判断一个对象的类是否实现了某个接口。



# 接口的实现

一个类要实现接口时，要注意下列问题：

- 在类的声明部分，用**implements**关键字声明该类将实现那些接口。
- 如果实现了某个接口的类不是**abstract**的抽象类，则在类的定义部分必须实现指定接口的所有抽象方法，即为所有抽象方法定义方法体。
- 如果实现了某个接口的类是**abstract**的抽象类，则它可以不实现指定接口的所有抽象方法。



# 接口的实现

- 接口的抽象方法的访问控制符为**public**，所以类在实现方法时，必须显式地使用**public**。
- 实现接口的类要实现接口的全部方法。如果不需要某个方法，也要定义成一个空方法体的方法。

如：

```
public 方法名() { ..... }
```

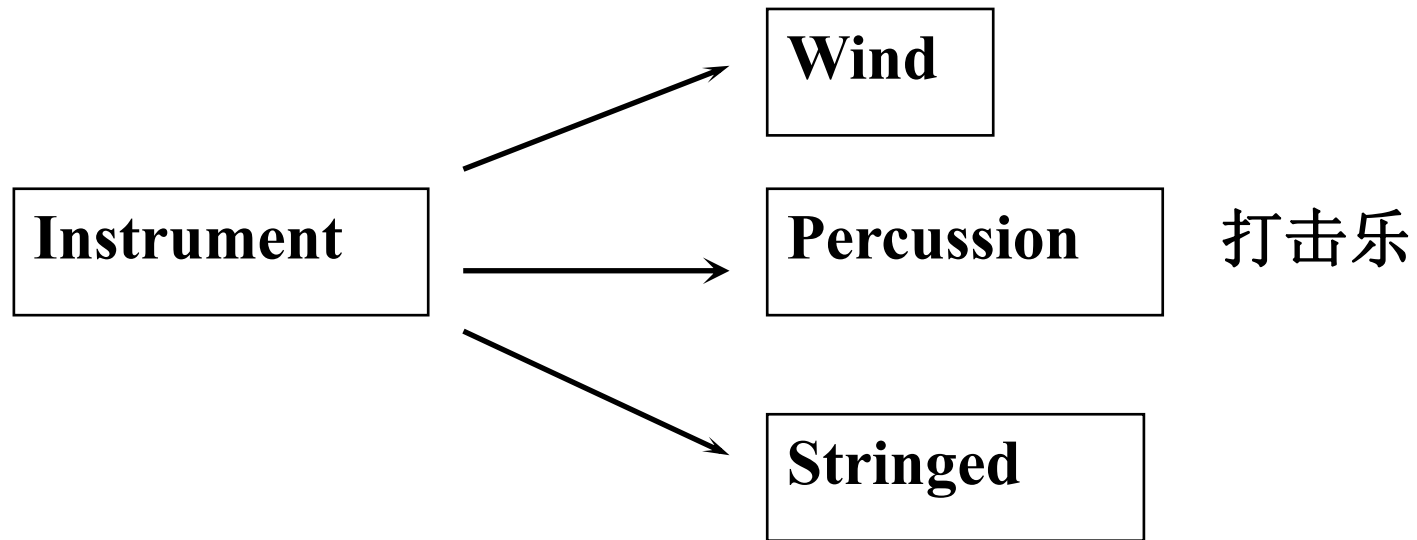


# 接口示例

```
interface CalArea {  
    double pi = 3.14 ;  
    double cal(double r) ;  
}  
class ExA implements CalArea {  
    public double cal(double r) {  
        return pi * r *r ;  
    }  
}
```

# 接口 (interface)

➤ 例：乐器族谱Music.java



我们可以使用接口来实现继承吗？

```
interface Instrument {  
    // constant:  
    int i = 5 ; // static & final  
    // Cannot have method definitions:  
    void play(); // Automatically public  
    String what();  
    void adjust();  
}  
class Wind implements Instrument {  
    // public cannot be omitted  
    public void play() {  
        System.out.println("Wind.play()");  
    }  
    public String what() { return "Wind"; }  
    //注意"adjust(){} "与"adjust();"的区别  
    public void adjust() {}  
}
```



# 接口的应用

- 声明多个类必须实现的方法。
  - ✓ 编程者可以把用于完成特定功能的方法组织成相对独立的集合体—接口。凡是需要实现这种特定功能的类，都必须实现这个接口。
- 利用接口模拟多继承。
  - ✓ **java**程序中的类层次结构是树状结构，这种树状结构在处理某些复杂问题时会显得力不从心。
  - ✓ 接口在保持单继承优点的前提下，使**Java**程序的类层次结构更加合理，更符合实际情况。
- 只说明对象的编程接口，而不揭示实际的类体。
  - ✓ 类库分发





# 系统定义的接口

```
public interface ActionListener extends EventListener
{
    public abstract void actionPerformed(ActionEvent e);
}
```

- 该接口代表了监听并处理动作事件的功能，它包含了一个抽象方法 `actionPerformed()`。
- 所有希望能够处理动作事件（如单击按钮、在文本框中回车等）的类都必须具有 `ActionListener` 接口定义的功能，也就是要实现这个接口

2020-10-21 AI Wen  
28日上机

## 4.10 内部类 (inner classes)

- 在类体中定义的新的类，称为inner classes



主要应用在事件驱动的控制结构中，如Java1.2的GUI应用

内部类的作用

# 实例1： 内部类的基本结构

//外部类

**class Out {**

private int age = 22;

//内部类

**class In {**

public void print() {

System.out.println(age);

}

}

//

}

//请区别并行定义的不是内部类

**public class Demo {**

public static void main(String[] args) {

Out.In in = new Out().new In();

in.print();

//或者采用下种方式访问

/\*

Out out = new Out();

Out.In in = out.new In();

in.print();

\*/

}

}



# 内部类 (inner class)

- 在某个类的内部定义的类称之为内部类。
- 内部类的定义方法
  - ✓ 定义命名的内部类：可以在类中（甚至方法中）定义内部类，并在类的内部多次使用（创建多个对象）。
  - ✓ 定义匿名内部类（一次性使用）：可以在new关键字后定义内部类，并立即创建一个对象
- 内部类的类文件命名方法

设外层类名为Myclass，则该类的内部类名为：

Myclass\$c1.class (c1为命名内部类名)

Myclass\$1.class (表示类中定义的第一个匿名内部类)

# 类中的内部类

- 在类中如同使用其他类一样使用自己的内部类。
- 内部类拥有对在外层类中定义的所有属性和方法的访问权。
- 其他类如果要使用一个类的内部类时，要使用类前缀。

从上面的例子不难看出，内部类其实严重破坏了良好的代码结构，但为什么还要使用内部类呢？

因为内部类可以随意使用外部类的成员变量（包括私有）而不用生成外部类的对象，这也是内部类的唯一优点。



# 类中的内部类

- 一般类只能是public和非public，而内部类可以指定为 **private** 和 **protected**。
- 如果内部类为**private**，只有本类可以使用它。
- 如果内部类为**protected**，只有外层类、与外层类处于同一包中的类、以及外层类的子类可以访问它。
- 内部类可以实现接口及抽象类中的抽象方法。

# 方法中的内部类

```
class Out {  
    private int age = 12;  
  
    public void Print(final int x) {  
        class In {  
            public void inPrint() {  
                System.out.println(x);  
                System.out.println(age);  
            }  
        }  
        new In().inPrint();  
    }  
}
```

```
public class Demo {  
    public static void  
    main(String[] args) {  
        Out out = new Out();  
        out.Print(3);  
    }  
}
```



# 方法中的内部类

- 内部类还可以定义在一个方法里，其作用域仅限于该方法的范围内(进一步隐藏),在其它方法里定义也没有名字冲突问题。
- ✓ 实现一个接口
- ✓ 在一编写好的代码中加一个类，但又不想公开化。

**StaticInnerClassTest.java**

**/CHP1-3/**





# 实现接口的无名内部类

```
return new Contents() {           //调用隐含构造函数
    private int i = 11;           //Contents是接口
    public int value() { return i; }
};                                //分号表示表达式的结束
```

该语句将返回值同表示返回值类型的类定义结合在一起。它相当于：

```
class MyContents implements Contents {
    private int i = 11;
    public int value() { return i; }
}
return new MyContents();
```



# 有名内部类例

```
public class MyFrame extends Frame {  
    ...  
    button1.addActionListener(new Button1Adapter());  
    ...  
    class Button1Adapter implements ActionListener { !  
        public void actionPerformed(ActionEvent e) {  
            ...  
        }  
    } // end of Button1Adapter class  
} // end of MyFrame class
```

常见

CS 2020-10-27 WEN 二

# 匿名内部类

## ➤ 匿名内部类也就是没有名字的内部类

- ✓ 正因为没有名字，所以匿名内部类只能使用一次，它通常用来简化代码编写。
- ✓ 但使用匿名内部类还有个前提条件：必须继承一个父类或实现一个接口。

➤ 匿名类可实例化一个接口 `button1.addActionListener(new ActionListener() {`  
    `public void actionPerformed(ActionEvent e) {`  
        `....`  
    `}`  
`};`);

- 使用接口创建对象的一般途径是创建实现该接口的一个类定义，然后使用类来创建一个对象。

# 内部类

## ➤ 例：Parcel.java

包裹

```
public class Parcel {  
    class Contents {  
        .....  
    }  
    class Destination {  
        .....  
    }  
    .....  
}
```

Parcel类中定义了两个inner classes

:

Contents  
和  
Destination

# 总结

- **final**可以用于类名前，表示类不可被继承；**final**用于变量前，表示它是只能一次赋值的变量，如果初始化了那就是常量。
- **static**可用于类内部的变量或方法前，表示这是整个类共享的变量和方法，与**类对象**无关。
- **abstract**用于类名前表示一个**抽象类**，**abstract**用于成员方法前表示**抽象方法**，而**抽象类**内部至少要有一个**抽象方法**，这个跟**final**不能一起用，至于为什么，你一想就清楚了，**抽象类**本身必须有子类需要实现它的方法，但是**final类**又不能有子类，这不矛盾了么，所以不能一起用。
- **static final** 用于修饰类的**静态变量**时表示一个常数，其实更多的还是用在接口里，毕竟用在类里面可能会被子类隐藏

# 课后作业

- 1.举例说明protected方法和友好方法的区别。
- 2.举例说明类方法和实例方法以及类变量和实例变量的区别。
- 3.子类将继承父类的哪些成员变量和方法？子类在什么情况下隐藏父类的成员变量和方法？在子类中是否允许有一个方法和父类的方法名字相同，而类型不同？说明你的理由？
- 4.下列程序有什么错误？

```
public class Takecare
{ int a=90; //
  static float b=10.98f;
  public static void main(String args[])
  { float c=a+b; //
    System.out.println("="+c);
  }
}
```

# 求梅森素数

- 所谓梅森数，是指形如 $2^p-1$ 的一类数，其中指数 $p$ 是素数，常记为 $M_p$ 。如果梅森数是素数，就称为梅森素数。
- 用因式分解法可以证明，若 $2^n-1$ 是素数，则指数 $n$ 也是素数；反之，当 $n$ 是素数时， $2^n-1$ （即 $M_p$ ）却未必是素数。前几个较小的梅森数大都是素数，然而梅森数越大，梅森素数也就越难出现。
- 目前仅发现51个梅森素数，最大的是 $M_{82589933}$ （即2的82589933次方减1），有24862048位数。

$$\begin{aligned}2^2-1 &= 3 \\2^3-1 &= 7 \\2^5-1 &= 31 \\2^7-1 &= 127 \\2^{13}-1 &= 8,191 \\2^{17}-1 &= 131,071 \\2^{19}-1 &= 524,287\end{aligned}$$



# 课后作业

5.上机编译运行下列程序。

```
class 学生
{ String 书,笔;int 学号,年级;
  学生(int number,int grade )
  { 学号=number;年级=grade;
  }
  void 去教室()
  { System.out.println("我带着"+书+"和"+笔+"来到了教室，准备听课");
  }
}

class Univ
{ public static void main(String args[])
  { 学生 张小林=new 学生(9901,2);
    张小林.书="英语书"; 张小林.笔="钢笔"; 张小林.去教室();
  }
}
```