



# 类的继承

李玮玮

# 讲授思路

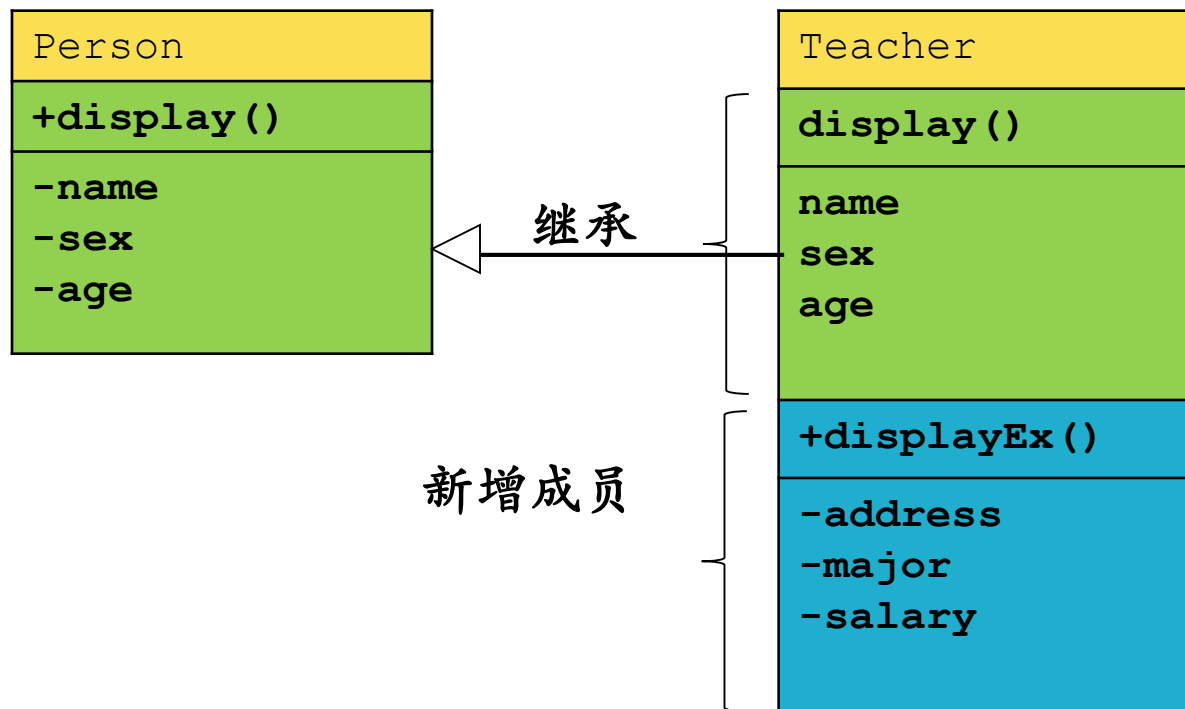
- 继承的实现
- 方法重写
- 抽象类和接口
- Object类

# 继承的实现

- 继承的概念
- 继承的语法
- 继承举例
- 构造方法的调用
- super关键字

# 继承的概念

- 继承就是从已有的类(父类)产生一个新的子类，子类通过继承自动拥有父类的非私有的属性和方法，继承是实现类的重用、软件复用的重要手段。





# 继承的语法

- 继承的语法格式：
  - class 子类名称 extends 父类名称{
    - //扩充或修改的属性与方法;
  - }
- 要点
  - Java 中的单继承机制：**一个类只能有一个直接父类;**
  - final修饰的类不能有子类;String是典型特例
  - Object类是所有Java类的顶级父类。
- Eclipse中安装JD反编译插件（参考网址）
  - <http://jingyan.baidu.com/article/fc07f9896da51512ffe5198a.html>
  - <http://blog.csdn.net/faithmy509/article/details/44494313>

# 继承举例

```
class Person {  
    private String name;  
    private String sex;  
    private int age;  
    //set和get方法省略  
    public void display(){  
    }  
}
```

```
class Teacher extends Person {  
    private String address;  
    private String major;  
    private double salary;  
    //set和get方法省略  
    public void displayEx(){  
    }  
}
```



```
class Person extends Object {  
    private String name;  
    private String sex;  
    private int age;  
    //set和get方法省略  
    public void display(){  
    }  
}
```

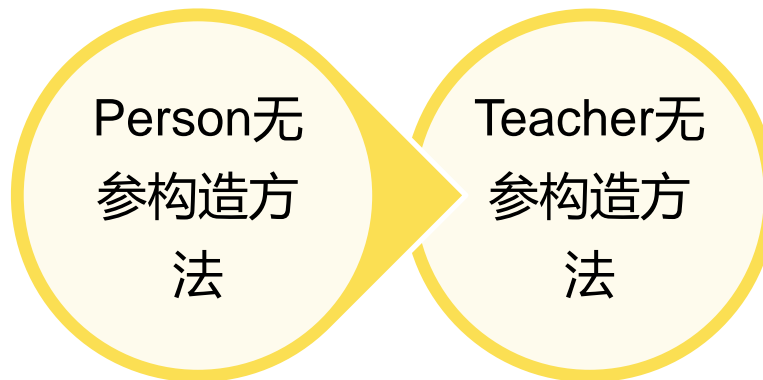
# 继承举例

```
public class Test{  
    public static void main(String[] args) {  
        Person p1 = new Person ();  
        Teacher t1 = new Teacher();  
        p1.display();  
        t1.displayEx();  
        // t1.display();  
    }  
}
```

- 问题1：t1.display();能否被正确执行？
- 问题2：通过语句Teacher t1 = new Teacher();创建对象时，是否执行父类的构造方法？

# 构造方法的调用

- 从本质上讲，实例化子类对象时系统会先调用父类的构造方法，然后再调用子类的构造方法。
  - JVM默认会调用父类中无参数的构造方法，若父类中没有无参数的构造方法，程序会报错。





# 构造方法的调用

- 思考：如何解决？
  - 方法一：在父类中定义无参数的构造方法。
  - 方法二：在子类中显示调用父类中定义的构造方法。

# 构造方法的调用

```
class Person {  
    private String name;  
    private String sex;  
    private int age;  
    //set和get方法省略  
    public Person(String name){  
        .....  
    }  
    public Person(){  
        .....  
    }  
    public void display(){  
    }  
}
```



```
class Teacher extends Person {  
    private String address;  
    private String major;  
    private double salary;  
    //set和get方法省略  
    public Teacher(){  
        super("张老师");  
        .....  
    }  
    public void displayEx(){  
    }  
}
```



# super关键字

- super用于引用父类中的属性或方法
  - super.属性、super.方法()
  - 注意：子类只能访问父类中的非private（私有）类型的属性或方法。

```
class Person{
    private String name;
    public void display(){ }
}
class Teacher extends Person{
    public void display(){
        super.name="teacher"; //编译器报错
        super.display();
    }
}
```

# super关键字

- 调用父类的构造方法
  - 使用super()、super(参数)的形式。
  - super()调用只能放在子类构造方法的第一行。
  - 子类构造方法中没有显示调用父类构造方法，则默认调用父类的无参构造方法。

# super关键字总结

- 调用父类的属性
  - 只能访问父类非private类型的属性
  - 使用 super.属性名称 的形式，位置不固定
- 调用父类的方法
  - super()或super(参数)调用父类的构造方法，只能放在子类构造方法的第一行。
  - 子类构造方法中没有显示调用父类构造方法，则默认调用父类的无参构造方法。
  - super.方法名([参数]) 调用父类非private类型的普通方法，位置不固定
- 区别于this关键字的使用

# 方法的重写

- 方法重写概念
- 方法重写的规则
- 方法重写的意义
- 多态的概念
- 多态的实现

# 方法重写

- 在继承机制中子类继承了父类的属性和方法，同时还可以对父类中的属性和方法做进一步的扩充或者修改；子类对父类中声明（定义）的方法进行重新实现的改造称为方法重写（override）。

```
class Person {  
    private String name;  
    public void display() {  
        System.out.println("Person display");  
    }  
}  
class Teacher extends Person{  
    public void display(){  
        System.out.println("Teacher display");  
    }  
}
```

# 方法重写

```
public class TestOverride {  
    public static void main (String[] args) {  
        Person person = new Person();  
        Teacher teacher = new Teacher();  
        person.display();  
        teacher.display();  
    }  
}  
  
class Person{  
    private String name;  
    public void display() {  
        System.out.println ("Person display");  
    }  
}  
  
class Teacher extends Person {  
}
```

输出结果：

Person display

Person display



# 方法重写

- 重写只能出现在继承关系之中。当一个类继承它的父类方法时，都有机会重写该父类的方法。重写的主要优点是能够定义某个子类型特有的行为。
- 对于从父类继承来的抽象方法，要么在子类用重写的方式设计该方法，要么把子类也标识为抽象的。所以抽象方法可以说是必须要被重写的方法。

# 方法重写的规则

- 子类中可以对父类中定义的方法进行改造，但必须遵循一定的规则：
  - 重写的方法返回类型一致;
  - 重写的方法具有相同的方法名;
  - 重写的方法参数列表必须相同;
  - 不能重写被标识为final的方法;
  - 重写的方法不能缩小访问权限;
  - 如果一个方法不能被继承，则不能重写它。如：父类的private方法。

# 方法重写的意义

- 方法重写最大的作用就是在不改变原来代码的基础上可以对其中任意模块进行改造。
- 举例：

```
class Person {  
    private String name;  
    public void display() {  
        System.out.println("Person display");  
    }  
}  
  
class Teacher extends Person {  
    public void display() {  
        System.out.println("override");  
        System.out.println("Teacher display");  
    }  
}
```

# 方法重写的意义

- 重写方法可以实现多态，用父类的引用来操纵子类对象，但是在实际运行中对象将运行其自己特有的方法。
- 父类的引用可以指向子类的对象：向上转型
- 好处
  - 可以使用子类强大的功能
  - 可以抽取父类的共性
- 指向子类的对象的父类类型的引用可以调用父类中定义属性和方法
- 指向子类的对象的父类类型的引用无法调用子类中定义而父类中没有的方法

# 抽象类和接口

- 抽象类的概念
- 抽象类的继承
- 接口的概念
- 接口的定义
- 接口的实现
- 接口的继承

# 抽象类的概念

- `abstract`修饰的类称为抽象类，抽象类的特点：
  - 不能实例化对象;
  - 类中可以定义抽象方法（`abstract`修饰的方法）;
  - 抽象类中可以没有抽象方法。
- `abstract`修饰的方法称为抽象方法，抽象方法只有方法的声明没有方法实现，即没有方法体。包含抽象方法的类本身必须被声明为抽象的。

```
abstract class Animal {  
    private String color ;  
    public abstract void shout();  
}
```

# 抽象类的继承

- 子类继承抽象类必须实现抽象类中所有的抽象方法，否则子类也必须定义为抽象类。

```
class Cat extends Animal {  
    public void shout() {  
        System.out.println("喵喵喵~~~");  
    }  
}
```

```
abstract class Cat extends Animal {  
}
```

```
class PersiaCat extends Cat {  
    public void shout() {  
        System.out.println("波斯猫喵~~~");  
    }  
}
```

# 接口的概念

- Java中的接口是一系列方法的声明，是一些方法特征的集合，接口可以看做是一种特殊的抽象类，其中包含常量和方法的声明，而没有变量和方法的实现。



# 接口的定义

- 接口的定义语法：
  - interface 接口名称 {
    - //接口中的常量声明
    - //接口中的抽象方法声明
  - }
- 举例：

```
interface Comparable {  
    int compareTo(Object other);  
}
```

# 接口的实现

- 类可以通过实现接口的方式来具有接口中定义的功能，基本语法：
  - class 类名 implements 接口名 {
    - .....
  - }
- 要点
  - 一个类可以同时实现多个接口;
  - 一个接口可以被多个无关的类实现;
  - 一个类实现接口必须实现接口中所有的抽象方法，否则必须定义为抽象类。

```
class Employee implements Comparable {  
    public int compareTo(Object other) {  
        ...  
    }  
}
```

# 接口的继承

- Java中接口可以继承接口，与类的继承概念一致，一个接口继承一个父接口就会继承父接口中定义的所有方法和属性。
- Java中接口的继承是多继承机制，即一个接口可以同时继承多个接口。
- 接口继承的基本语法：
  - interface 接口名 extends 父接口1,父接口2,.....{
    - .....
  - }

```
interface interFaceA extends interFace1,interFace2 {  
    //接口的其他代码  
}
```

# 接口的意义

- 弥补Java中单继承机制的不足。
- 接口只有方法的定义没有方法的实现，即都是抽象方法，这些方法可以在不同的地方被不同的类实现，而这些实现可以具有不同的行为（功能）。

# 抽象类和接口总结

- 抽象类
  - 不能实例化，但是可以声明抽象类的引用
  - 包含抽象方法的类必须定义为抽象类
  - 不包含抽象方法的类不一定是抽象类（抽象类中可以不包含抽象方法）
  - 抽象方法不含方法体，必须显式定义为abstract（不同于void display()**{ }**）
  - 抽象类的子类必须实现基类的所有抽象方法，否则也必须定义为抽象类

# 抽象类和接口总结

- 接口
  - 弥补Java单一继承的不足
  - 不能实例化，但是可以声明接口变量
  - 包含常量和方法的声明，不含变量和方法的实现（常量可以没有final修饰，必须初始化，在接口实现类中不能修改值；方法可以没有abstract，在接口实现类中必须实现）
  - 接口中的抽象方法不能有方法体（即便是空的方法体{ }也不行）
  - 接口不能继承类
  - 接口可以继承（extends）1个或多个接口（,分隔开）
  - 一个类可以实现（implements）1个或多个接口（,分隔开）

# Object类

- Object类概述
- 方法预览
- 方法使用说明

# Object类概述

- Object类是所有Java类的祖先。每个类都使用 Object 作为超类。所有对象（包括数组）都实现这个类的方法。
- 在不明确给出超类的情况下，Java会自动把Object作为要定义类的超类。
- 可以使用类型为Object的变量指向任意类型的对象。



# Object类概述

- Object类有一个默认构造方法 `public Object()`，在构造子类实例时，都会先调用这个默认构造方法。
- Object类的变量只能用作各种值的通用持有者。要对他们进行任何专门的操作，都需要知道它们的原始类型并进行类型转换。
- 例如：
  - `Object obj = new MyObject();`
  - `MyObject x = (MyObject) obj;`

# 方法预览

- `Object()`
  - 默认构造方法。
- `clone()`
  - 创建并返回此对象的一个副本。
- `equals(Object obj)`
  - 指示某个其他对象是否与此对象“相等”。
- `finalize()`
  - 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
- `getClass()`
  - 返回一个对象的运行时类。

# 方法预览

- hashCode()
  - 返回该对象的哈希码值。
- notify()
  - 唤醒在此对象监视器上等待的单个线程。
- notifyAll()
  - 唤醒在此对象监视器上等待的所有线程。
- toString()
  - 返回该对象的字符串表示。
- wait()
  - 导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。

# 方法预览

- `wait(long timeout)`
  - 导致当前的线程等待，直到其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法，或者超过指定的时间量。
- `wait(long timeout, int nanos)`
  - 导致当前的线程等待，直到其他线程调用此对象的 `notify()` 方法或 `notifyAll()` 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量。

# 方法使用说明

- equals方法
- hashCode方法
- toString方法

# equals方法

- equals()方法：用于测试某个对象是否同另一个对象相等。它在Object类中的实现是判断两个对象是否指向同一块内存区域。
- 这种测试用处不大，因为即使内容相同的对象，内存区域也是不同的。如果想测试对象是否相等，就需要覆盖此方法，进行更有意义的比较。

# equals方法

- 举例：如果两个雇员对象的姓名、薪水、雇佣日期都一样，就认为这两个对象是相等的。

```
class Employee{
    public boolean equals(Object otherObj){
        //快速测试是否是同一个对象
        if(this == otherObj) return true;
        //如果显式参数为null，必须返回false
        if(otherObj == null) return false;
        //如果类不匹配，就不可能相等
        if(getClass() != otherObj.getClass()) return false;
        //现在已经知道otherObj是个非空的Employee对象
        Employee other = (Employee)otherObj;
        //测试所有的字段是否相等
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}
```

# equals方法

- Java语言规范要求equals方法具有下面的特点：
  - 自反性：对于任何非空引用值  $x$ ， $x.equals(x)$  都应返回 `true`。
  - 对称性：对于任何非空引用值  $x$  和  $y$ ，当且仅当  $y.equals(x)$  返回 `true` 时， $x.equals(y)$  才应返回 `true`。
  - 传递性：对于任何非空引用值  $x$ 、 $y$  和  $z$ ，如果  $x.equals(y)$  返回 `true`，并且  $y.equals(z)$  返回 `true`，那么  $x.equals(z)$  应返回 `true`。
  - 一致性：对于任何非空引用值  $x$  和  $y$ ，多次调用  $x.equals(y)$  始终返回 `true` 或始终返回 `false`，前提是对象上 `equals` 比较中所用的信息没有被修改。
  - 对于任何非空引用值  $x$ ， $x.equals(null)$  都应返回 `false`。



# hashCode方法

- `public int hashCode()` 返回该对象的哈希码值。
- hashCode 的常规协定是：
  - 在 Java 应用程序执行期间，在同一对象上多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是对象上 equals 比较中所用的信息没有被修改。
  - 如果根据 equals(Object) 方法，两个对象是相等的，那么在两个对象中的每个对象上调用 hashCode 方法都必须生成相同的整数结果。

# toString方法

- `public String toString()` 返回该对象的字符串表示。
- 通常，`toString` 方法会返回一个“以文本方式表示”此对象的字符串。建议所有子类都重写此方法。
- `Object` 类的 `toString` 方法返回一个字符串，该字符串由类名（对象是该类的一个实例）、`at` 标记符“@”和此对象哈希码的无符号十六进制表示组成。
- 该方法返回一个字符串，它的值等于：
  - `getClass().getName() + '@' + Integer.toHexString(hashCode())`

# 总结

- 继承的概念
- 继承的语法
- 继承举例
- 构造方法的调用
- super关键字

# 课后阅读

- this 与 super 的区别。



**Thank You**