

1.Java继承

继承是java面向对象编程技术的一块基石，因为它允许创建分等级层次的类。
继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。

类的继承格式:

```
class 父类 {  
}  
class 子类 extends 父类 {  
}
```

接下来我们通过实例来说明这个需求。

开发动物类，其中动物分别为企鹅以及老鼠，要求如下：

企鹅：属性（姓名，id），方法（吃，睡，自我介绍）
老鼠：属性（姓名，id），方法（吃，睡，自我介绍）

企鹅类：

```
public class Penguin {  
    private String name;  
    private int id;  
    public Penguin(String myName, int myid) {  
        name = myName;  
        id = myid;  
    }  
    public void eat(){  
        System.out.println(name+"正在吃");  
    }  
    public void sleep(){  
        System.out.println(name+"正在睡");  
    }  
    public void introduction() {  
        System.out.println("大家好! 我是" + id + "号" + name + ".");  
    }  
}
```

老鼠类：

```
public class Mouse {  
    private String name;  
    private int id;  
    public Mouse(String myName, int myid) {  
        name = myName;  
        id = myid;  
    }  
    public void eat(){
```

```

        System.out.println(name+"正在吃");
    }
    public void sleep(){
        System.out.println(name+"正在睡");
    }
    public void introduction() {
        System.out.println("大家好! 我是" + id + "号" + name + ".");
    }
}

```

从这两段代码可以看出来，代码存在重复了，导致后果就是代码量大且臃肿，而且维护性不高（维护性主要是后期需要修改的时候，就需要修改很多的代码，容易出错），所以要从根本上解决这两段代码的问题，就需要继承，将两段代码中相同的部分提取出来组成一个父类：

公共父类：

```

public class Animal {
    private String name;
    private int id;
    public Animal(String myName, int myid) {
        name = myName;
        id = myid;
    }
    public void eat(){
        System.out.println(name+"正在吃");
    }
    public void sleep(){
        System.out.println(name+"正在睡");
    }
    public void introduction() {
        System.out.println("大家好! 我是" + id + "号" + name + ".");
    }
}

```

这个 **Animal** 类就可以作为一个父类，然后企鹅类和老鼠类继承这个类之后，就具有父类当中的属性和方法，子类就不会存在重复的代码，维护性也提高，代码也更加简洁，提高代码的复用性（复用性主要是可以多次使用，不用再多次写同样的代码）继承之后的代码：

企鹅类：

```

public class Penguin extends Animal {
    public Penguin(String myName, int myid) {
        super(myName, myid);
    }
}

```

老鼠类：

```

public class Mouse extends Animal {
    public Mouse(String myName, int myid) {
        super(myName, myid);
    }
}

```

继承的特性

子类拥有父类非 **private** 的属性、方法。

子类可以拥有自己的属性和方法，即子类可以对父类进行扩展。

子类可以用自己的方式实现父类的方法。

Java 的继承是单继承，但是可以多重继承，单继承就是一个子类只能继承一个父类，多重继承就是，例如 A 类继承 B 类，B 类继承 C 类，所以按照关系就是 C 类是 B 类的父类，B 类是 A 类的父类，这是 Java 继承区别于 C++ 继承的一个特性。

提高了类之间的耦合性（继承的缺点，耦合度高就会造成代码之间的联系越紧密，代码独立性越差）。

继承关键字

继承可以使用 **extends** 和 **implements** 这两个关键字来实现继承，而且所有的类都是继承于 **java.lang.Object**，当一个类没有继承的两个关键字，则默认继承 **Object**（这个类在 **java.lang** 包中，所以不需要 **import**）祖先类。

extends关键字

在 **Java** 中，类的继承是单一继承，也就是说，一个子类只能拥有一个父类，所以 **extends** 只能继承一个类。

```
public class Animal {
    private String name;
    private int id;
    public Animal(String myName, String myid) {
        //初始化属性值
    }
    public void eat() { //吃东西方法的具体实现 }
    public void sleep() { //睡觉方法的具体实现 }
}

public class Penguin extends Animal{
}
```

implements关键字

使用 **implements** 关键字可以变相的使**java**具有多继承的特性，使用范围为类继承接口的情况，可以同时继承多个接口（接口跟接口之间采用逗号分隔）

```
public interface A {
    public void eat();
    public void sleep();
}

public interface B {
    public void show();
}

public class C implements A,B {
}
```

super 与 this 关键字

super关键字：我们可以通过**super**关键字来实现对父类成员的访问，用来引用当前对象的父类。

this关键字：指向自己的引用。

```
class Animal {
    void eat() {
        System.out.println("animal : eat");
    }
}

class Dog extends Animal {
    void eat() {
        System.out.println("dog : eat");
    }
    void eatTest() {
        this.eat();    // this 调用自己的方法
        super.eat();   // super 调用父类方法
    }
}

public class Test {
    public static void main(String[] args) {
        Animal a = new Animal();
        a.eat();
        Dog d = new Dog();
        d.eatTest();
    }
}
```

输出结果为：

animal : eat

dog : eat

animal : eat

final关键字

final 关键字声明类可以把类定义为不能继承的，即最终类；或者用于修饰方法，该方法不能被子类重写

```
声明类：
final class 类名 { //类体}

声明方法：
修饰符(public/private/default/protected) final 返回值类型 方法名(){ //方法体}
```

注:实例变量也可以被定义为 **final**，被定义为 **final** 的变量不能被修改。被声明为 **final** 类的方法自动地声明为 **final**，但是实例变量并不是 **final**

构造器

子类是不继承父类的构造器（构造方法或者构造函数）的，它只是调用（隐式或显式）。如果父类的构造器带有参数，则必须在子类的构造器中显式地通过 **super** 关键字调用父类的构造器并配以适当的参数列表。

如果父类构造器没有参数，则在子类的构造器中不需要使用 `super` 关键字调用父类构造器，系统会自动调用父类的无参构造器。

```
class SuperClass {
    private int n;
    SuperClass(){
        System.out.println("SuperClass()");
    }
    SuperClass(int n) {
        System.out.println("SuperClass(int n)");
        this.n = n;
    }
}
// SubClass 类继承
class SubClass extends SuperClass{
    private int n;

    SubClass(){ // 自动调用父类的无参数构造器
        System.out.println("SubClass");
    }

    public SubClass(int n){
        super(300); // 调用父类中带有参数的构造器
        System.out.println("SubClass(int n):"+n);
        this.n = n;
    }
}
// SubClass2 类继承
class SubClass2 extends SuperClass{
    private int n;

    SubClass2(){
        super(300); // 调用父类中带有参数的构造器
        System.out.println("SubClass2");
    }

    public SubClass2(int n){ // 自动调用父类的无参数构造器
        System.out.println("SubClass2(int n):"+n);
        this.n = n;
    }
}
public class TestSuperSub{
    public static void main (String args[]){
        System.out.println("-----SubClass 类继承-----");
        SubClass sc1 = new SubClass();
        SubClass sc2 = new SubClass(100);
        System.out.println("-----SubClass2 类继承-----");
        SubClass2 sc3 = new SubClass2();
        SubClass2 sc4 = new SubClass2(200);
    }
}
```

结果：

```
-----SubClass 类继承-----
SuperClass()
SubClass
SuperClass(int n)
SubClass(int n):100
-----SubClass2 类继承-----
SuperClass(int n)
SubClass2
SuperClass()
SubClass2(int n):200
```

Java 重写(Override)与重载(Overload)

重写(Override)

重写是子类对父类的允许访问的方法的实现过程进行重新编写, 返回值和形参都不能改变。即外壳不变，核心重写！

重写的好处在于子类可以根据需要，定义特定于自己的行为。也就是说子类能够根据需要实现父类的方法。

重写方法不能抛出新的检查异常或者比被重写方法申明更加宽泛的异常。例如：父类的一个方法申明了一个检查异常 `IOException`，但是在重写这个方法的时候不能抛出 `Exception` 异常，因为 `Exception` 是 `IOException` 的父类，只能抛出 `IOException` 的子类异常。

```
class Animal{
    public void move(){
        System.out.println("动物可以移动");
    }
}

class Dog extends Animal{
    public void move(){
        System.out.println("狗可以跑和走");
    }
}

public class TestDog{
    public static void main(String args[]){
        Animal a = new Animal(); // Animal 对象
        Animal b = new Dog(); // Dog 对象

        a.move();// 执行 Animal 类的方法

        b.move();//执行 Dog 类的方法
    }
}
```

以上实例编译运行结果如下：

```
动物可以移动
狗可以跑和走
```

在上面的例子中可以看到，尽管**b**属于**Animal**类型，但是它运行的是**Dog**类的**move**方法。

这是由于在编译阶段，只是检查参数的引用类型。

然而在运行时，**Java**虚拟机(**JVM**)指定对象的类型并且运行该对象的方法。

因此上面的例子中，之所以能编译成功，是因为**Animal**类中存在**move**方法，然而运行时，运行的是特定对象的方法。

思考以下例子：

```
class Animal{
    public void move(){
        System.out.println("动物可以移动");
    }
}

class Dog extends Animal{
    public void move(){
        System.out.println("狗可以跑和走");
    }
}
```

```

    }
    public void bark(){
        System.out.println("狗可以吠叫");
    }
}

public class TestDog{
    public static void main(String args[]){
        Animal a = new Animal(); // Animal 对象
        Animal b = new Dog(); // Dog 对象

        a.move();// 执行 Animal 类的方法
        b.move();//执行 Dog 类的方法
        b.bark();
    }
}

```

以上实例编译运行结果如下：

```

TestDog.java:30: cannot find symbol
symbol   : method bark()
location: class Animal
        b.bark();

```

该程序将抛出一个编译错误，因为b的引用类型Animal没有bark方法。

方法的重写规则

参数列表必须完全与被重写方法的相同； 返回类型与被重写方法的返回类型可以不相同，但是必须是父类返回值的派生类（**java5**及更早版本返回类型要一样，**java7**及更高版本可以不同）； 访问权限不能比父类中被重写的方法的访问权限更低。例如：如果父类的一个方法被声明为**public**，那么在子类中重写该方法就不能声明为**protected**。父类的成员方法只能被它的子类重写。声明为**final**的方法不能被重写。声明为**static**的方法不能被重写，但是能够被再次声明。子类和父类在同一个包中，那么子类可以重写父类所有方法，除了声明为**private**和**final**的方法。子类和父类不在同一个包中，那么子类只能够重写父类的声明为**public**和**protected**的非**final**方法。重写的方法能够抛出任何非强制异常，无论被重写的方法是否抛出异常。但是，重写的方法不能抛出新的强制性异常，或者比被重写方法声明的更广泛的强制性异常，反之则可以。构造方法不能被重写。如果不能继承一个方法，则不能重写这个方法。

Super关键字的使用

当需要在子类中调用父类的被重写方法时，要使用**super**关键字。

```

class Animal{
    public void move(){
        System.out.println("动物可以移动");
    }
}

class Dog extends Animal{
    public void move(){
        super.move(); // 应用super类的方法
        System.out.println("狗可以跑和走");
    }
}

public class TestDog{
    public static void main(String args[]){

```

```

        Animal b = new Dog(); // Dog 对象
        b.move(); //执行 Dog类的方法

    }
}

```

以上实例编译运行结果如下：

```

动物可以移动
狗可以跑和走

```

重载(Overload)

重载(overloading) 是在一个类里面，方法名字相同，而参数不同。返回类型可以相同也可以不同。

每个重载的方法（或者构造函数）都必须有一个独一无二的参数类型列表。

最常用的地方就是构造器的重载。

重载规则:

被重载的方法必须改变参数列表(参数个数或类型不一样);

被重载的方法可以改变返回类型;

被重载的方法可以改变访问修饰符;

被重载的方法可以声明新的或更广的检查异常;

方法能够在同一个类中或者在一个子类中被重载。 无法以返回值类型作为重载函数的区分标准。

```

public class Overloading {
    public int test(){
        System.out.println("test1");
        return 1;
    }

    public void test(int a){
        System.out.println("test2");
    }

    //以下两个参数类型顺序不同
    public String test(int a,String s){
        System.out.println("test3");
        return "returntest3";
    }

    public String test(String s,int a){
        System.out.println("test4");
        return "returntest4";
    }

    public static void main(String[] args){
        Overloading o = new Overloading();
        System.out.println(o.test());
        o.test(1);
        System.out.println(o.test(1,"test3"));
        System.out.println(o.test("test4",1));
    }
}

```

重写与重载之间的区别

区别点	重载方法	重写方法
参数列表	必须修改	一定不能修改
返回类型	可以修改	一定不能修改
异常	可以修改	可以减少或删除，一定不能抛出新的或者更广的异常
访问	可以修改	一定不能做更严格的限制（可以降低限制）

总结

方法的重写(Overriding)和重载(Overloading)是java多态性的不同表现，重写是父类与子类之间多态性的一种表现，重载可以理解成多态的具体表现形式。

(1)方法重载是一个类中定义了多个方法名相同,而他们的参数的数量不同或数量相同而类型和次序不同,则称为方法的重载(Overloading)。(2)方法重写是在子类存在方法与父类的方法的名字相同,而且参数的个数与类型一样,返回值也一样的方法,就称为重写(Overriding)。(3)方法重载是一个类的多态性表现,而方法重写是子类与父类的一种多态性表现。

Overriding 重写

```

class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}

```

方法名与参数都一样

Overloading 重载

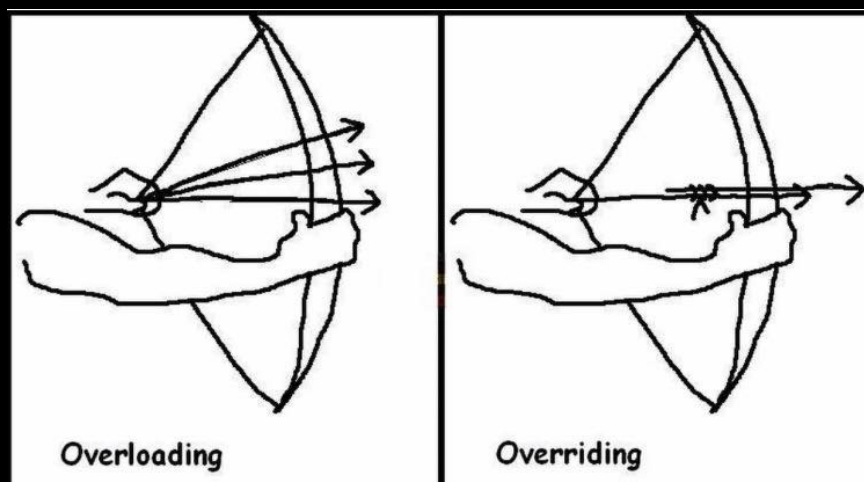
```

class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}

```

方法名相同，参数不同



Java 多态

多态是同一个行为具有多个不同表现形式或形态的能力。

多态就是同一个接口，使用不同的实例而执行不同操作，如图所示：

多态的优点

1. 消除类型之间的耦合关系
2. 可替换性
3. 可扩充性
4. 接口性
5. 灵活性
6. 简化性

多态存在的三个必要条件

继承

重写 父类引用指向子类对象

当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

多态的好处：可以使程序有良好的扩展，并可以对所有类的对象进行通用处理。

```
public class Test {
    public static void main(String[] args) {
        show(new Cat()); // 以 Cat 对象调用 show 方法
        show(new Dog()); // 以 Dog 对象调用 show 方法

        Animal a = new Cat(); // 向上转型
        a.eat();               // 调用的是 Cat 的 eat
        Cat c = (Cat)a;        // 向下转型
        c.work();              // 调用的是 Cat 的 work
    }

    public static void show(Animal a) {
        a.eat();
        // 类型判断
        if (a instanceof Cat) { // 猫做的事情
            Cat c = (Cat)a;
            c.work();
        } else if (a instanceof Dog) { // 狗做的事情
            Dog c = (Dog)a;
            c.work();
        }
    }
}

abstract class Animal {
    abstract void eat();
}

class Cat extends Animal {
    public void eat() {
        System.out.println("吃鱼");
    }
    public void work() {
        System.out.println("抓老鼠");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("吃骨头");
    }
    public void work() {
        System.out.println("看家");
    }
}
```

```
}
```

执行以上程序，输出结果为：

```
吃鱼
抓老鼠
吃骨头
看家
吃鱼
抓老鼠
```

虚函数

虚函数的存在是为了多态。

Java 中其实没有虚函数的概念，它的普通函数就相当于 C++ 的虚函数，动态绑定是 Java 的默认行为。如果 Java 中不希望某个函数具有虚函数特性，可以加上 **final** 关键字变成非虚函数。

重写

我们将介绍在 Java 中，当设计类时，被重写的方法的行为怎样影响多态性。

我们已经讨论了方法的重写，也就是子类能够重写父类的方法。

当子类对象调用重写的方法时，调用的是子类的方法，而不是父类中被重写的方法。

要想调用父类中被重写的方法，则必须使用关键字 **super**。

```
Employee.java 文件代码：
/* 文件名 : Employee.java */
public class Employee {
    private String name;
    private String address;
    private int number;
    public Employee(String name, String address, int number) {
        System.out.println("Employee 构造函数");
        this.name = name;
        this.address = address;
        this.number = number;
    }
    public void mailCheck() {
        System.out.println("邮寄支票给: " + this.name
            + " " + this.address);
    }
    public String toString() {
        return name + " " + address + " " + number;
    }
    public String getName() {
        return name;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String newAddress) {
        address = newAddress;
    }
}
```

```

    public int getNumber() {
        return number;
    }
}

```

假设下面的类继承Employee类:

```

/* 文件名 : Salary.java */
public class Salary extends Employee
{
    private double salary; // 全年工资
    public Salary(String name, String address, int number, double salary) {
        super(name, address, number);
        setSalary(salary);
    }
    public void mailCheck() {
        System.out.println("Salary 类的 mailCheck 方法 ");
        System.out.println("邮寄支票给: " + getName()
            + " , 工资为: " + salary);
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double newSalary) {
        if(newSalary >= 0.0) {
            salary = newSalary;
        }
    }
    public double computePay() {
        System.out.println("计算工资, 付给: " + getName());
        return salary/52;
    }
}

```

尝试给出它的输出结果:

```

/* 文件名 : VirtualDemo.java */
public class VirtualDemo {
    public static void main(String [] args) {
        Salary s = new Salary("员工 A", "北京", 3, 3600.00);
        Employee e = new Salary("员工 B", "上海", 2, 2400.00);
        System.out.println("使用 Salary 的引用调用 mailCheck -- ");
        s.mailCheck();
        System.out.println("\n使用 Employee 的引用调用 mailCheck--");
        e.mailCheck();
    }
}

```

以上实例编译运行结果如下:

```

Employee 构造函数
Employee 构造函数
使用 Salary 的引用调用 mailCheck --
Salary 类的 mailCheck 方法
邮寄支票给: 员工 A , 工资为: 3600.0

使用 Employee 的引用调用 mailCheck--
Salary 类的 mailCheck 方法
邮寄支票给: 员工 B , 工资为: 2400.0

```

例子解析

实例中，实例化了两个 **Salary** 对象：一个使用 **Salary** 引用 **s**，另一个使用 **Employee** 引用 **e**。

当调用 `s.mailCheck()` 时，编译器在编译时会在 **Salary** 类中找到 `mailCheck()`，执行过程 JVM 就调用 **Salary** 类的 `mailCheck()`。

因为 **e** 是 **Employee** 的引用，所以调用 **e** 的 `mailCheck()` 方法时，编译器会去 **Employee** 类查找 `mailCheck()` 方法。

在编译的时候，编译器使用 **Employee** 类中的 `mailCheck()` 方法验证该语句，但是在运行的时候，Java虚拟机(JVM)调用的是 **Salary** 类中的 `mailCheck()` 方法。

以上整个过程被称为虚拟方法调用，该方法被称为虚拟方法。

Java中所有的方法都能以这种方式表现，因此，重写的方法能在运行时调用，不管编译的时候源代码中引用变量是什么数据类型。

多态的实现方式

方式一：重写：这个内容已经在上一章节详细讲过，就不再阐述，详细可访问：[Java 重写\(Override\)与重载\(Overload\)](#)。

方式二：接口

1. 生活中的接口最具代表性的就是插座，例如一个三接头的插头都能接在三孔插座中，因为这个是每个国家都有各自规定的接口规则，有可能到国外就不行，那是因为国外自己定义的接口类型。
2. java中的接口类似于生活中的接口，就是一些方法特征的集合，但没有方法的实现。具体可以看 [java接口](#) 这一章节的内容。

方式三：抽象类和抽象方法

Java 抽象类

在面向对象的概念中，所有的对象都是通过类来描绘的，但是反过来，并不是所有的类都是用来描绘对象的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是抽象类。

抽象类除了不能实例化对象之外，类的其它功能依然存在，成员变量、成员方法和构造方法的访问方式和普通类一样。

由于抽象类不能实例化对象，所以抽象类必须被继承，才能被使用。也是因为这个原因，通常在设计阶段决定要不要设计抽象类。

父类包含了子类集合的常见的方法，但是由于父类本身是抽象的，所以不能使用这些方法。

在Java中抽象类表示的是一种继承关系，一个类只能继承一个抽象类，而一个类却可以实现多个接口。

抽象类

在Java语言中使用`abstract class`来定义抽象类。

```
/* 文件名 : Employee.java */
public abstract class Employee
{
    private String name;
    private String address;
    private int number;
    public Employee(String name, String address, int number)
    {
        System.out.println("Constructing an Employee");
        this.name = name;
        this.address = address;
        this.number = number;
    }
}
```

```

public double computePay()
{
    System.out.println("Inside Employee computePay");
    return 0.0;
}
public void mailCheck()
{
    System.out.println("Mailing a check to " + this.name
        + " " + this.address);
}
public String toString()
{
    return name + " " + address + " " + number;
}
public String getName()
{
    return name;
}
public String getAddress()
{
    return address;
}
public void setAddress(String newAddress)
{
    address = newAddress;
}
public int getNumber()
{
    return number;
}
}

```

注意到该 `Employee` 类没有什么不同，尽管该类是抽象类，但是它仍然有 3 个成员变量，7 个成员方法和 1 个构造方法。现在如果你尝试如下的例子：

```

/* 文件名 : AbstractDemo.java */
public class AbstractDemo
{
    public static void main(String [] args)
    {
        /* 以下是不允许的，会引发错误 */
        Employee e = new Employee("George W.", "Houston, TX", 43);

        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

```

当你尝试编译 `AbstractDemo` 类时，会产生如下错误：

```

Employee.java:46: Employee is abstract; cannot be instantiated
    Employee e = new Employee("George W.", "Houston, TX", 43);
                  ^
1 error

```

继承抽象类

我们能够通过一般的方法继承 `Employee` 类：

```

/* 文件名 : Salary.java */
public class Salary extends Employee
{
    private double salary; //Annual salary
    public Salary(String name, String address, int number, double
        salary)
    {
        super(name, address, number);
        setSalary(salary);
    }
    public void mailCheck()
    {
        System.out.println("Within mailCheck of Salary class ");
        System.out.println("Mailing check to " + getName()
            + " with salary " + salary);
    }
    public double getSalary()
    {
        return salary;
    }
    public void setSalary(double newSalary)
    {
        if(newSalary >= 0.0)
        {
            salary = newSalary;
        }
    }
    public double computePay()
    {
        System.out.println("Computing salary pay for " + getName());
        return salary/52;
    }
}

```

尽管我们不能实例化一个 `Employee` 类的对象，但是如果我们实例化一个 `Salary` 类对象，该对象将从 `Employee` 类继承 7 个成员方法，且通过该方法可以设置或获取三个成员变量。

```

/* 文件名 : AbstractDemo.java */
public class AbstractDemo
{
    public static void main(String [] args)
    {
        Salary s = new Salary("Mohd Mohtashim", "Ambehta, UP", 3, 3600.00);
        Employee e = new Salary("John Adams", "Boston, MA", 2, 2400.00);

        System.out.println("Call mailCheck using Salary reference --");
        s.mailCheck();

        System.out.println("\n Call mailCheck using Employee reference--");
        e.mailCheck();
    }
}

```

以上程序编译运行结果如下：

```

Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

```

```
Call mailCheck using Employee reference--  
Within mailCheck of Salary class  
Mailing check to John Adams with salary 2400.
```

抽象方法

如果你想设计这样一个类，该类包含一个特别的成员方法，该方法的具体实现由它的子类确定，那么你可以在父类中声明该方法为抽象方法。

Abstract 关键字同样可以用来声明抽象方法，抽象方法只包含一个方法名，而没有方法体。

抽象方法没有定义，方法名后面直接跟一个分号，而不是花括号。

```
public abstract class Employee  
{  
    private String name;  
    private String address;  
    private int number;  
  
    public abstract double computePay();  
  
    //其余代码  
}
```

声明抽象方法会造成以下两个结果：

如果一个类包含抽象方法，那么该类必须是抽象类。

任何子类必须重写父类的抽象方法，或者声明自身为抽象类。

继承抽象方法的子类必须重写该方法。否则，该子类也必须声明为抽象类。最终，必须有子类实现该抽象方法，否则，从最初的父类到最终子类都不能用来实例化对象。

如果**Salary**类继承了**Employee**类，那么它必须实现**computePay()**方法

```
/* 文件名 : Salary.java */  
public class Salary extends Employee  
{  
    private double salary; // Annual salary  
  
    public double computePay()  
    {  
        System.out.println("Computing salary pay for " + getName());  
        return salary/52;  
    }  
  
    //其余代码  
}
```

抽象类总结规定

1. 抽象类不能被实例化(初学者很容易犯的错)，如果被实例化，就会报错，编译无法通过。只有抽象类的非抽象子类可以创建对象。
2. 抽象类中不一定包含抽象方法，但是有抽象方法的类必定是抽象类。
3. 抽象类中的抽象方法只是声明，不包含方法体，就是不给出方法的具体实现也就是方法的具体功能。

4. 构造方法，类方法（用 **static** 修饰的方法）不能声明为抽象方法。
5. 抽象类的子类必须给出抽象类中的抽象方法的具体实现，除非该子类也是抽象类。

Java 封装

在面向对象程式设计方法中，封装（英语：**Encapsulation**）是指一种将抽象性函式接口的实现细节部份包装、隐藏起来的方法。

封装可以被认为是一个保护屏障，防止该类的代码和数据被外部类定义的代码随机访问。

要访问该类的代码和数据，必须通过严格的接口控制。

封装最主要的功能在于我们能修改自己的实现代码，而不用修改那些调用我们代码的程序片段。

适当的封装可以让程式码更容易理解与维护，也加强了程式码的安全性。

封装的优点

1. 良好的封装能够减少耦合。
2. 类内部的结构可以自由修改。
3. 可以对成员变量进行更精确的控制。
4. 隐藏信息，实现细节。

实现Java封装的步骤

1. 修改属性的可见性来限制对属性的访问（一般限制为**private**），例如：

```
public class Person {  
    private String name;  
    private int age;  
}
```

这段代码中，将 **name** 和 **age** 属性设置为私有的，只能本类才能访问，其他类都访问不了，如此就对信息进行了隐藏。

2. 对每个值属性提供对外的公共方法访问，也就是创建一对赋取值方法，用于对私有属性的访问，

```
public class Person{  
    private String name;  
    private int age;  
  
    public int getAge(){  
        return age;  
    }  
  
    public String getName(){  
        return name;  
    }  
  
    public void setAge(int age){  
        this.age = age;  
    }  
}
```

```

    }

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

```

采用 **this** 关键字是为了解决实例变量（**private String name**）和局部变量（**setName(String name)**中的**name**变量）之间发生的同名的冲突。

```

/* 文件名: EncapTest.java */
public class EncapTest{

    private String name;
    private String idNum;
    private int age;

    public int getAge(){
        return age;
    }

    public String getName(){
        return name;
    }

    public String getIdNum(){
        return idNum;
    }

    public void setAge( int newAge){
        age = newAge;
    }

    public void setName(String newName){
        name = newName;
    }

    public void setIdNum( String newId){
        idNum = newId;
    }
}

```

以上实例中**public**方法是外部类访问该类成员变量的入口。

通常情况下，这些方法被称为**getter**和**setter**方法。

因此，任何要访问类中私有成员变量的类都要通过这些**getter**和**setter**方法。

通过如下的例子说明**EncapTest**类的变量怎样被访问：

```

/* F文件名 : RunEncap.java */
public class RunEncap{
    public static void main(String args[]){
        EncapTest encap = new EncapTest();
        encap.setName("James");
        encap.setAge(20);
        encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName()+

```

```
        " Age : "+ encap.getAge());  
    }  
}
```

以上代码编译运行结果如下:

```
Name : James Age : 20
```

Java 接口

接口（英文：**Interface**），在**JAVA**编程语言中是一个抽象类型，是抽象方法的集合，接口通常以**interface**来声明。一个类通过继承接口的方式，从而来继承接口的抽象方法。

接口并不是类，编写接口的方式和类很相似，但是它们属于不同的概念。类描述对象的属性和方法。接口则包含类要实现的方法。

除非实现接口的类是抽象类，否则该类要定义接口中的所有方法。

接口无法被实例化，但是可以被实现。一个实现接口的类，必须实现接口内所描述的所有方法，否则就必须声明为抽象类。另外，在 Java 中，接口类型可用来声明一个变量，他们可以成为一个空指针，或是被绑定在一个以此接口实现的对象。

接口与类相似点:

一个接口可以有多个方法。

接口文件保存在 **.java** 结尾的文件中，文件名使用接口名。

接口的字节码文件保存在 **.class** 结尾的文件中。

接口相应的字节码文件必须在与包名称相匹配的目录结构中。

接口与类的区别:

接口不能用于实例化对象。接口没有构造方法。接口中所有的方法必须是抽象方法。接口不能包含成员变量，除了 **static** 和 **final** 变量。接口不是被类继承了，而是要被类实现。接口支持多继承。

接口特性

接口中每一个方法也是隐式抽象的,接口中的方法会被隐式的指定为 **public abstract**（只能是 **public abstract**，其他修饰符都会报错）。

接口中可以含有变量，但是接口中的变量会被隐式的指定为 **public static final** 变量（并且只能是 **public**，用 **private** 修饰会报编译错误）。

接口中的方法是不能在接口中实现的，只能由实现接口的类来实现接口中的方法。

抽象类和接口的区别

1. 抽象类中的方法可以有方法体，就是能实现方法的具体功能，但是接口中的方法不行。
2. 抽象类中的成员变量可以是各种类型的，而接口中的成员变量只能是 **public static final** 类型的。

3. 接口中不能含有静态代码块以及静态方法(用 **static** 修饰的方法)，而抽象类是可以有静态代码块和静态方法。

4. 一个类只能继承一个抽象类，而一个类却可以实现多个接口。

接口的声明语法格式如下：

```
/* 文件名 : NameOfInterface.java */
import java.lang.*;
//引入包

public interface NameOfInterface
{
    //任何类型 final, static 字段
    //抽象方法
}
}
```

接口有以下特性：

接口是隐式抽象的，当声明一个接口的时候，不必使用**abstract**关键字。

接口中每一个方法也是隐式抽象的，声明时同样不需要**abstract**关键字。

接口中的方法都是公有的。

```
/* 文件名 : Animal.java */
interface Animal {
    public void eat();
    public void travel();
}
```

接口的实现

当类实现接口的时候，类要实现接口中所有的方法。否则，类必须声明为抽象的类。

类使用**implements**关键字实现接口。在类声明中，**Implements**关键字放在**class**声明后面。

实现一个接口的语法，可以使用这个公式：

...**implements** 接口名称 [, 其他接口名称, 其他接口名称..., ...] ...

```
public class MammalInt implements Animal{

    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }

    public int noOfLegs(){
        return 0;
    }
}
```

```
public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
}
}
```

以上实例编译运行结果如下:

```
Mammal eats
Mammal travels
```

重写接口中声明的方法时，需要注意以下规则：

类在实现接口的方法时，不能抛出强制性异常，只能在接口中，或者继承接口的抽象类中抛出该强制性异常。

类在重写方法时要保持一致的方法名，并且应该保持相同或者相兼容的返回值类型。

如果实现接口的类是抽象类，那么就没必要实现该接口的方法。

在实现接口的时候，也要注意一些规则：

一个类可以同时实现多个接口。

一个类只能继承一个类，但是能实现多个接口。

一个接口能继承另一个接口，这和类之间的继承比较相似。

接口的继承

一个接口能继承另一个接口，和类之间的继承方式比较相似。接口的继承使用**extends**关键字，子接口继承父接口的方法。

下面的Sports接口被Hockey和Football接口继承：

```
// 文件名: Sports.java
public interface Sports
{
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}

// 文件名: Football.java
public interface Football extends Sports
{
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}

// 文件名: Hockey.java
public interface Hockey extends Sports
{
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
}
```

```
public void overtimePeriod(int ot);  
}
```

Hockey接口自己声明了四个方法，从**Sports**接口继承了两个方法，这样，实现**Hockey**接口的类需要实现六个方法。

相似的，实现**Football**接口的类需要实现五个方法，其中两个来自于**Sports**接口。

接口的多继承

在**Java**中，类的多继承是不合法，但接口允许多继承。

在接口的多继承中**extends**关键字只需要使用一次，在其后跟着继承接口。如下所示：

```
public interface Hockey extends Sports, Event
```

以上的程序片段是合法定义的子接口，与类不同的是，接口允许多继承，而 **Sports**及 **Event** 可能定义或是继承相同的方法

标记接口

最常用的继承接口是没有包含任何方法的接口。

标记接口是没有任何方法和属性的接口.它仅仅表明它的类属于一个特定的类型,供其他代码来测试允许做一些事情。

标记接口作用：简单形象的说就是给某个对象打个标（盖个戳），使对象拥有某个或某些特权。

例如：java.awt.event 包中的 **MouseListener** 接口继承的 java.util.EventListener 接口定义如下：

```
package java.util;  
public interface EventListener  
{}
```

没有任何方法的接口被称为标记接口。标记接口主要用于以下两种目的：

建立一个公共的父接口：

正如**EventListener**接口，这是由几十个其他接口扩展的**Java API**，你可以使用一个标记接口来建立一组接口的父接口。例如：当一个接口继承了**EventListener**接口，**Java虚拟机(JVM)**就知道该接口将要被用于一个事件的代理方案。

向一个类添加数据类型：

这种情况是标记接口最初的目的，实现标记接口的类不需要定义任何接口方法(因为标记接口根本就没有方法)，但是该类通过多态性变成一个接口类型。

Java 包(package)

包的作用

1、把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。

2、如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。

3、包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。

Java 使用包（package）这种机制是为了防止命名冲突，访问控制，提供搜索和定位类（class）、接口、枚举（enumerations）和注释（annotation）等。

例如,一个Something.java 文件它的内容

```
package net.java.util;
public class Something{
    ...
}
```

那么它的路径应该是 **net/java/util/Something.java** 这样保存的。**package(包)** 的作用是把不同的 **java** 程序分类保存,更方便的被其他 **java** 程序调用。

一个包 (**package**) 可以定义为一组相互联系的类型 (类、接口、枚举和注释), 为这些类型提供访问保护和命名空间管理的功能。

以下是一些 Java 中的包:

```
java.lang-{} 包基础类
java.io-{} 包含输入输出功能的包
```

开发者可以自己把一组类和接口等打包,并定义自己的包。而且在实际开发中这样做是值得提倡的,当你自己完成类的实现之后,将相关的类分组,可以让其他的编程者更容易地确定哪些类、接口、枚举和注释等是相关的。

由于包创建了新的命名空间 (**namespace**), 所以不会跟其他包中的任何名字产生命名冲突。使用包这种机制,更容易实现访问控制,并且让定位相关类更加简单。

创建包

让我们来看一个例子,这个例子创建了一个叫做**animals**的包。通常使用小写的字母来命名避免与类、接口名字的冲突。在**animals**包中加入一个接口 (**interface**):

```
package animals;

interface Animal {
    public void eat();
    public void travel();
}
```

接下来,在同一个包中加入该接口的实现:

```
MammalInt.java 文件代码:
package animals;

/* 文件名 : MammalInt.java */
public class MammalInt implements Animal{

    public void eat(){
        System.out.println("Mammal eats");
    }

    public void travel(){
        System.out.println("Mammal travels");
    }
}
```

```

    public int noOfLegs(){
        return 0;
    }

    public static void main(String args[]){
        MammalInt m = new MammalInt();
        m.eat();
        m.travel();
    }
}

```

然后，编译这两个文件，并把他们放在一个叫做`animals`的子目录中。用下面的命令来运行：

```

$ mkdir animals
$ cp Animal.class MammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travel

```

import 关键字

为了能够使用某一个包的成员，我们需要在 **Java** 程序中明确导入该包。使用 **"import"** 语句可完成此功能。

在 **java** 源文件中 **import** 语句应位于 **package** 语句之后，所有类的定义之前，可以没有，也可以有多条，其语法格式为：

```
import package1[.package2...].(classname|*);
```

如果在一个包中，一个类想要使用本包中的另一个类，那么该包名可以省略。

下面的 `payroll` 包已经包含了 `Employee` 类，接下来向 `payroll` 包中添加一个 `Boss` 类。`Boss` 类引用 `Employee` 类的时候可以不用使用 `payroll` 前缀，`Boss`类的实例如下。

```

package payroll;

public class Boss
{
    public void payEmployee(Employee e)
    {
        e.mailCheck();
    }
}

```

如果 `Boss` 类不在 `payroll` 包中又会怎样？`Boss` 类必须使用下面几种方法之一来引用其他包中的类。

使用类全名描述，例如

```
payroll.Employee
```

用 `import` 关键字引入，使用通配符 `"*"`

```
import payroll.*;
```


使用 `import` 关键字引入 `Employee` 类:

```
import payroll.Employee;
```

注意:

类文件中可以包含任意数量的 **import** 声明。**import** 声明必须在包声明之后，类声明之前。

package 的目录结构

包名成为类名的一部分，正如我们前面讨论的一样。

包名必须与相应的字节码所在的目录结构相吻合。

下面是管理你自己 `java` 中文件的一种简单方式:

将类、接口等类型的源码放在一个文本中，这个文件的名称就是这个类型的名称，并以 `.java` 作为扩展名。例如:

```
// 文件名 : Car.java

package vehicle;

public class Car {
    // 类实现
}
```

接下来，把源文件放在一个目录中，这个目录要对应类所在包的名称。

```
....\vehicle\Car.java
```

现在，正确的类名和路径将会是如下样子:

类名 -> `vehicle.Car`

路径名 -> `vehicle\Car.java` (在 windows 系统中)

通常，一个公司使用它互联网域名的颠倒形式来作为它的包名。例如: 互联网域名是 `XXX.com`，所有的包名都以 `com.xxx` 开头。包名中的每一个部分对应一个子目录。

例如: 有一个 `com.runoob.test` 的包，这个包包含一个叫做 `Runoob.java` 的源文件，那么相应的，应该有如下面的一连串子目录:

```
....\com\runoob\test\Runoob.java
```

编译的时候，编译器为包中定义的每个类、接口等类型各创建一个不同的输出文件，输出文件的名称就是这个类型的名称，并加上 `.class` 作为扩展后缀。例如:

```
// 文件名: Runoob.java

package com.runoob.test;
public class Runoob {

}

class Google {
```

```
}
```

现在，我们用-d选项来编译这个文件，如下：

```
$javac -d . Runoob.java
```

这样会像下面这样放置编译了的文件：

```
.\com\runoob\test\Runoob.class  
.\com\runoob\test\Google.class
```

你可以像下面这样来导入所有 \com\runoob\test\ 中定义的类、接口等：

```
import com.runoob.test.*;
```

编译之后的 .class 文件应该和 .java 源文件一样，它们放置的目录应该跟包的名字对应起来。但是，并不要求 .class 文件的路径跟相应的 .java 的路径一样。你可以分开来安排源码和类的目录。

```
<path-one>\sources\com\runoob\test\Runoob.java  
<path-two>\classes\com\runoob\test\Google.class
```

这样，你可以将你的类目录分享给其他的编程人员，而不用透露自己的源码。用这种方法管理源码和类文件可以让编译器和java 虚拟机（JVM）可以找到你程序中使用的所有类型。

类目录的绝对路径叫做 class path。设置在系统变量 CLASSPATH 中。编译器和 java 虚拟机通过将 package 名字加到 class path 后来构造 .class 文件的路径。

\classes 是 class path，package 名字是 com.runoob.test,而编译器和 JVM 会在 \classes\com\runoob\test 中找 .class 文件。

一个 class path 可能会包含好几个路径，多路径应该用分隔符分开。默认情况下，编译器和 JVM 查找当前目录。JAR 文件按包含 Java 平台相关的类，所以他们的目录默认放在了 class path 中。

设置 CLASSPATH 系统变量 用下面的命令显示当前的CLASSPATH变量：

```
Windows 平台（DOS 命令行下）：C:\> set CLASSPATH  
UNIX 平台（Bourne shell 下）：# echo $CLASSPATH  
删除当前CLASSPATH变量内容：  
Windows 平台（DOS 命令行下）：C:\> set CLASSPATH=  
UNIX 平台（Bourne shell 下）：# unset CLASSPATH; export CLASSPATH  
设置CLASSPATH变量：  
  
Windows 平台（DOS 命令行下）：C:\> set CLASSPATH=C:\users\jack\java\classes  
UNIX 平台（Bourne shell 下）：# CLASSPATH=/home/jack/java/classes; export CLASSPATH
```

1 text

2 text

3 text

4 text

5 text

7 text

8 text

9 text

0 text

01 text

02 text