1.如何实现类似函数指针的功能

**2.组合与继承的区别**

**2.1．继承**

名为Insect（昆虫）的类，包含两方法：1）移动move()； 2）攻击attack()。

|  |
| --- |
| class Insect {  private int size;  private String color;  public Insect(int size, String color) {  this.size = size;  this.color = color;  }  public int getSize() {  return size;  }  public void setSize(int size) {  this.size = size;  }  public String getColor() {  return color;  }  public void setColor(String color) {  this.color = color;  }  public void move() {  System.out.println("Move");  }  public void attack() {  move(); //假设昆虫在攻击前必须要先移动一次  System.out.println("Attack");  }  } |

现想要定义名为Bee（蜜蜂）的类。

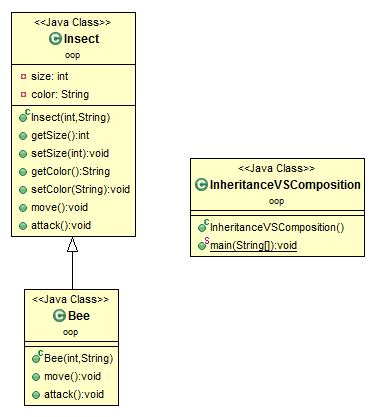
Bee（蜜蜂）是Insect（昆虫）的一种，但实现了不同于Insect（昆虫）的attack()和move方法。

可用继承来实现Bee类：

|  |
| --- |
| class Bee extends Insect {  public Bee(int size, String color) {  super(size, color);  }  public void move() {  System.out.println("Fly");  }  public void attack() {  move();  super.attack();//造成Fly被打印了两次  }  }    public class InheritanceVSComposition {  public static void main(String[] args) {  Insect i = new Bee(1, "red");  i.attack();  }  } |

InheritanceVSComposition作为一个测试类，在其main方法中生成了一个Bee类的实例，并赋值给Insect类型的引用变量 i。所以调用i的attack方法时，对应的是Bee类实例的attack方法，也就是调用了Bee类的attack方法。

类的继承结构图如下



Fly

Fly

Attack

Fly被打印了两次，也就是说move方法被调用了两次。但按理来讲，move方法只应当被调用一次，因为无论是昆虫还是蜜蜂，一次攻击前只移动一次。

问题出在子类（即Bee类）的attack方法的重载代码中，也就是super.attack()这一句。因为在父类（即Insect类）中，调用 attack方法时会先调用move方法，所以当子类（Bee）调用super.attack()时，相当于也同时调用了被重写的move方法（注意是子 类被重载的move方法，而不是父类的move方法）。

为解决这个问题，可采取以下办法：

删除子类的attack方法。这么做会使子类的attack方法的实现完全依赖于父类对于该方法的实现（因为子类继承了父类的attack方法）。如父类的attack方法不受控制而产生了变更。比如说，父类的attack方法中调用了另外的move方法，那么子类的attack方法也会产生相应的变 化，是一种很糟糕的封装。

也可重写子类的attack方法，像下面这样：

|  |
| --- |
| public void attack() {  move();  System.out.println("Attack");  } |

这样保证了结果的正确性，因为子类的attack方法不再依赖于父类。

但子类attack方法的代码与父类产生了重复（重复的attack方法会使得很多事情变得复杂，不仅仅是多打印了一条输出语句）。所以第二种办法也不行，它不符合软件工程中关于重用的思想。

如此看来，继承机制是有缺点的：子类依赖于父类的实现细节，如果父类产生了变更，子类的后果将不堪设想。

**2.1 组合**

可用组合机制来替代继承。

attack这一功能被抽象为一个接口。

|  |
| --- |
| interface Attack {  public void move();  public void attack();  } |

通过对Attack接口的实现，就可在实现类中定义不同类型的attack。

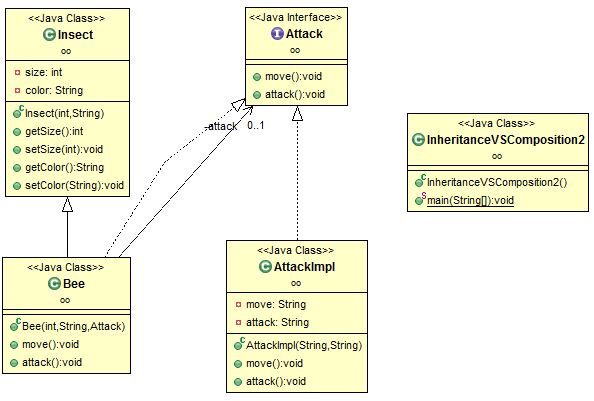
|  |
| --- |
| class AttackImpl implements Attack {  private String move;  private String attack;    public AttackImpl(String move, String attack) {  this.move = move;  this.attack = attack;  }    @Override  public void move() {  System.out.println(move);  }    @Override  public void attack() {  move();  System.out.println(attack);  }  } |

因attack功能已被抽象为一个接口，所以Insect类不再需要有attack方法。

|  |
| --- |
| class Insect {  private int size;  private String color;    public Insect(int size, String color) {  this.size = size;  this.color = color;  }    public int getSize() {  return size;  }    public void setSize(int size) {  this.size = size;  }    public String getColor() {  return color;  }    public void setColor(String color) {  this.color = color;  }  } |

Bee是一种Insect类，具有attack的功能，所以它实现了attack接口：

|  |
| --- |
| // 这个封装类封装了一个Attack类型的对象  class Bee extends Insect implements Attack {  private Attack attack;    public Bee(int size, String color, Attack attack) {  super(size, color);  this.attack = attack;  }    public void move() {  attack.move();  }    public void attack() {  attack.attack();  }  } |



测试类代码:将AttackImpl的实例作为Attack类型的参数传给Bee类的构造函数：

|  |
| --- |
| public class InheritanceVSComposition2 {  public static void main(String[] args) {  Bee a = new Bee(1, "black", new AttackImpl("fly", "move"));  a.attack();    // if you need another implementation of move()  // there is no need to change Insect,  //we can quickly use new method to attack    Bee b = new Bee(1, "black", new AttackImpl("fly", "sting"));  b.attack();  }  } |

fly

move

fly

sting

3、什么时候用继承，什么时候用组合？

* 如存在一种IS-A的关系（比如Bee“是一个”Insect），且一个类需向另一个类暴露所有的方法接口，更应用继承机制。
* 如存在HAS-A的关系（比如Bee“有一个”attack功能），更应该用组合。

组合（对象持有）和继承都是实现系统功能重用，代码复用的最常用的有效的设计技巧，是设计模式中的基础结构。

4. 区别

* 区别1：
* 类继承允许根据自己的实现来覆盖重写父类的实现细节，父类的实现对于子类是可见的，一般称之为白盒复用。
* 对象持有（组合）要求建立一个好的接口，但整体类和部分类之间不会去关心各自的实现细节，即它们之间的实现细节不可见，故称为黑盒复用。
* 区别2：

继承是在编译时静态定义的，是静态复用，编译后子类和父类的关系就已确定了。

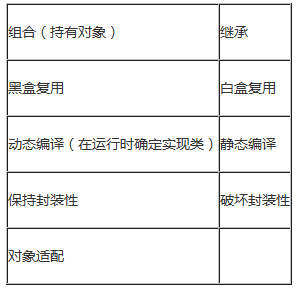
组合是运用于复杂的设计，关系在运行时才确定，即在对对象没有创建运行前，整体类是不会知道自己将持有特定接口下的哪个实现类。在扩展方面组合比继承更具有广泛性。

* 区别3：

继承中父类定义了子类的部分实现，而子类中又会重写这些实现，修改父类的实现，设计模式中认为这是一种破坏了父类的封装性的表现。导致父类实现的任何变化，必然导致子类的改变。然而组合这不会出现这种现象。

对象的组合还有一个优点就是有助于保持每个类被封装，并被集中在单个任务上（类设计的单一原则）。这样类的层次结构不会扩大，一般不会出现不可控的庞然大类。而类的继承就可能出来这些问题，所以一般编码规范都要求类的层次结构不要超过3层。组合是大型系统软件实现即插即用时的首选方式。

总结如下表



4. 具体使用如何选择?

“优先使用对象组合，而不是继承”是面向对象设计的第二原则。

理想情况下，不需创建新的组件来完成代码复用，而只需通过对象组合的方法来拼装已存在的组件以获取新的功能。但这种情况很少出现，因在实际情况中，现有的构建总是不够，而通过继承来复用代码往往比通过组合对象容易得多。

所以，继承和组合这两种方法并存于实际的软件开发过程中。

3.this域super的区别

4.assert左勇

5.volatile作用-多线程

6.strictfp作用

7.不可变类(immutable class)

8.字符串创建与存储的机制

9.== equals hashcode区别

10.异常处理的原理

11.java IO流的实现机制

12.java socket是啥

13.java NIO

14.序列化

15．GC

16.内存泄漏

**17.多线程**

使用的原因？如何实现？

多线程同步的实现方法

Sleep wait yield区别

终止线程的方法

守护线程

join作用

18.JDBC处理事务

19.jsp与servlet异同

20.如何用jsp与servlet实现MVC模型

21.ajax是啥

22.javabean EJB？

23.web服务器 web应用服务器

24.web service

25.SOAP 与 REST

26.什么是XML

27.数据库连接池的工作机制

28.spring IOC AOP

29.内连接、外连接

30．数据库事务

存储过程

范式

触发器

游标

视图

30.设计模式

工厂

适配器

观察者

Java中字符只以unicode存在(不选择任何特定编码，直接用它们在字符集中编号)：  
“java中”:JVM中、内存中、代码里声明的每一个char、string类型的变量中

1.i++ ++i

2.0+’0’ 整数+字符串

3.j=j++ java中间缓存变量机制

4.assert

5.main方法可不传参数？

6.三目运算符 右结合性 自动类型提升

运算符优先级

7.32>>32 移位运算 8位一循环，等价于32>>0

8.final finally finalize

package的作用

二．

1.java中传值还是传引用

|  |
| --- |
| 按值传递:当将一个参数传递给一个函数时，函数接收的是原始值的一个副本  按引用传递:当将一个参数传递给一个函数时，函数接收的是原始值的内存地址，而不是值的副本  1.对象就是传引用  2.原始类型就是传值  3.String类型因为没有提供自身修改的函数，每次操作都是新生成一个String对象，所以要特殊对待。可以认为是传值。 |

2.定义在类中的变量会被赋予默认值？static影响？

3.为何静态方法不能访问非静态变量

4. 

5.I/O流

6.序列化 （与持久化关系）

三.

1.递归计算斐波那契数列

2. 



（难）

3．利用1 2 3 4 5这5个·数字，打印出不同的排列，不能有重复

四．java内存管理

1.垃圾收集：优势 收集标准

2.java如何内存管理

3.内存泄漏：解释 引起的原因 哪些情况？

4.clone作用

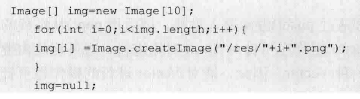
5.泛型与容器

UML

MVC理解:springmvc struts backbone

单例模式写法

5.未释放堆栈 空间原因：



6.栈操作：将1-12月的英文单词压入栈中，再将其取出，Java实现

(1) new stack()

(2)new vector()

7. 链表

8.hash表

9.树

先序 中序 后序遍历

平衡二叉树

10.图

图的深度优先遍历

图的广度优先遍历

哈夫曼编码问题

四叉树

Trie树，单词查找树，字典树

11.排序

稳定性

平均性能，最快的排序算法

有序/无序队列：寻找最小值/估算平均值/找出中间值/找出最大出现的可能性的时间复杂度

(b c d e f g q r s t)查找b，二分查找过程

写冒泡排序：输入10个数，输出排序结果

写选择拍寻：输入10个数，输出排序结果

写插入排序：输入10个数，输出排序结果

写希尔排序：输入10个数，输出排序结果

二分法排序思想？

理解快速排序代码

写归并排序：输入10个数，输出排序结果

辗转相除法，即欧几里得算法，时间复杂度？

数据库索引

1. 常见的5中RuntimeExecption

|  |
| --- |
| ClassCastException(类转换异常)：多态中，可以使用Instanceof 判断，进行规避  ArithmeticException("数学运算异常")：进行if判断，如果除数为0，进行return  NullPointerException：进行if判断，是否为null  ArrayIndexOutOfBoundsException：使用数组length属性，避免越界  IndexOutOfBoundsException(数组越界)  ClassCastException(类型强制转换异常) |

4.1.2

“Java语言中的方法属于类中的成员（member）”：不对

原因：类成员必须静态，而方法可能是静态方法，也可能是非静态方法。

静态方法是类成员，非静态方法是实例成员。

4.1.4 类被加载/类加载时机：静态代码块

|  |
| --- |
| **初始化规定：**   * 创建类的实例 * 访问类的静态变量 (除常量【被final修辞的静态变量】   原因:常量一种特殊的变量，因为编译器把他们当作值(value)而不是域(field)来对待。如果你的代码中用到了常变量(constant variable)，编译器并不会生成字节码来从对象中载入域的值，而是直接把这个值插入到字节码中。这是一种很有用的优化，但是如果你需要改变final域的值那么每一块用到那个域的代码都需要重新编译。   * 访问类的静态方法 * 反射 如( Class.forName(“my.xyz.Test”) ) * 当初始化一个类时，发现其父类还未初始化，则先出发父类的初始化 * 虚拟机启动时，定义了main()方法的那个类先初始化 |

1. 编译即javac过程，将.java文件compile成.class文件，主要是类型、格式检查与编译成字节码文件

加载是指java \*的过程，将.class文件加载到内存中去解释执行，即运行的时候才会有加载一说。

1. 类的加载时机，肯定是在运行时，但并不是一次性全部加载，而是按需动态，依靠反射来实现动态加载，一般来说一个class只会被加载一次，之后就会从jvm的class实例的缓存中获取，谁用谁取就可以了，不会再去文件系统中加载.class文件了。

|  |
| --- |
| 静态代码块在类加载时调用，并且只调用一次。  最常用的作用：给一个对象的属性初始化。  具体的说：  1.当调用一个类的静态变量时，类中的静态代码块会执行。【只有静态代码块会执行】  2.当调用一个类的静态方法时，类中的静态代码块会执行。【只有静态代码块会执行】  3.当创建一个类的实例时，类中的静态代码块、非静态代码块（也叫构造代码块）、创建实例的相应的构造方法都会执行。  调用顺序：静态代码块、非静态代码块、构造方法。 |

4.1.5 java程序初始化顺序

当实例化对象时，对象所在类的所有成员变量首先要初始化，之后才会调用对象所在类的构造函数创建对象。

初始化原则：

|  |
| --- |
| 1.静态对象/变量(只初始化一次)优先于非静态对象/变量(可能多次)初始化，   1. 父类优先于子类 2. 按成员变量定义顺序初始化。   及时变量定义分散了方法定义中，依旧在方法(含构造函数)调用前初始化。 |

初始化方式的执行顺序：

|  |
| --- |
| 父类静态变量、父类静态代码块、  子类静态变量、子类静态代码块  父类非静态变量、父类非静态代码块、父类构造函数  子类非静态变量、子类非静态代码块、子类构造函数 |

**4.1.6 java中作用域**

变量类型：

* **成员变量:**类实例化时，在内存中分配空间并初始化。与实例化对象生命周期相同
* **静态(全局)变量:**static修饰的成员变量。类加载时分配空间。
* **局部变量**



private:当前类有访问权限

protected：自己及其子类可见。

default：自己及位于同一包中的类可见。

注意:只能修饰成员变量。

private与protected不能修饰类。

Public、abstract、final可修饰类

**4.1.7 一个java文件是否可定义多个类**

可以。

但最多一个类被public修饰，且该类名需与文件名相同。

若文件中无public的类，文件名随便是一个类的名字即可。

编译时，会给每个类生成一个.class文件

**4.1.8构造函数**

在对象实例化时初始化对象的成员变量。

特点:

* 与类名相同，不能有返回值。
* 一个类可有多个构造函数。没写时，系统提供无参构造函数，写了就不提供。
* 随new一起被自动调用，不能由编写者直接调用。
* 不能被继承，所以不能被覆盖。能被重载
* 父类未提供无参构造函数,子类构造函数必须显示调用(super)父类构造函数

父类提供了，编译器会默认调用父类提供的无参构造函数。

当父类和子类都没定义构造函数，编译器会为他们生成…

* 默认构造函数的修饰符同当前类的修饰符。

**4.1.9 有些接口没有任何方法(标识接口)**

表明实现他的类属于某特定类型。

如Cloneable 、Serializable

使用时常用instanceof来判断实例对象类型是否实现了一个给定的标识接口(利用接口的多态性)。

**4.1.10 %% clone方法（book）**

* 浅复制:被复制变量的变量复制，引用的对象仍指向原对象。
* 深复制:被复制对象的变量和引用的对象都复制。

**4.1.11 反射机制**

允许程序在运行时自我检查，对其内部成员进行操作。

能实现运行时对类进行加载，因此可增加程序的灵活性，当不恰当的使用会严重影响系统性能。

* **功能:**

|  |
| --- |
| 得到一个对象所属的类  获取类的成员变量和方法  在运行时创建对象  在运行时调用对象的方法 |

* 例子:

|  |
| --- |
| package javastudy.reflect;  class Base {  public void f() {  System.out.println("Base");  }  }  class Sub extends Base {  @Override  public void f() {  System.out.println("Sub");  }  }  public class Test {  public static void main(String[] args) {  try {  //使用反射机制加载类  //Class c = Class.forName("Sub");//报错:找不到  Class c = Class.forName("javastudy.reflect.Sub");  //Object newInstance = c.newInstance();  Base b = (Base)c.newInstance();  b.f();  } catch (Exception e) {  e.printStackTrace();  }  }  } |

结果:打印”Sub”

* 反射机制获取class类:

|  |
| --- |
| Class.forName(“类的路径”)  类名.class  实例.getClass |

Java创建对象的4种方法：

|  |
| --- |
| new对象  反射机制创建(3种)  Clone方法  反序列化 |