1.如何实现类似函数指针的功能

**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

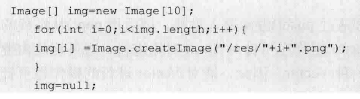
5.泛型与容器

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：

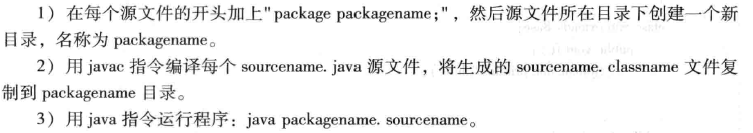
|  |
| --- |
| new对象  反射机制创建(3种)  Clone方法  反序列化 |

**4.1.12 package**

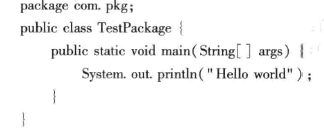
* **作用:**

1. 提供多层命名空间，解决命名冲突。使得不同package中的类可同名
2. 对类按功能分类，使项目组织清晰。

* **用法:**



例子:



javac –d TestPackage.java编译代码，会在当前目录下生成com/pkg目录

Java com.pkg.TestPackage执行程序

**4.1.13 如何实现类似函数指针的功能(p62) (没仔细看)**

策略设计模式思想

**4.2 面向对象技术**

**4.2.1 面向对象与面向过程区别(p65) (没仔细看)**

**4.2.2 面向对象的特征**

* **抽象**：注意与当前目标有关的方面。过程和数据抽象
* **封装**：将客观事物抽象成类，类对自身的数据和方法实行保护。可只让可信的类或对象操作。
* **继承：**表述共性。派生类可从基类继承方法和实例变量，也可修改或增加新的方法。
* **多态：**灵活、抽象、行为共享、代码共享优势

**4.2.3 面向对象开发方式优点**

* 较高开发效率:把现实事物直接映射为开发对象。可通过继承或组合方式实现代码重用。提高开发效率。
* 保证软件鲁棒性:因其高的重用性
* 保证软件的高可维护性:代码可读性好，结构清晰。成熟的设计模式。

**4.2.4 继承**

特性:  
1.单继承

2.子类只能继承父类非私有(public protected)的成员变量和方法

1. 成员变量同名时不会继承（成员变量无多态性）

父类和子类的成员变量同时存在，即使同名，互相隐藏。

子类中看到的是子类的变量，父类看到的是父类中变量。

|  |  |
| --- | --- |
| 重写和重载针对方法，子类的变量可覆盖父类的变量，但不能改变父类的变量。   |  | | --- | | package javastudy.reflect;  class Animals {  int age = 10;  void enjoy() {  System.out.println("Animals enjoy!");  }    }    class Dogg extends Animals {  int age = 20;  int weight;  void enjoy() {  System.out.println("Dog enjoy!");  }  }    public class TestDuoTai {  public static void main(String[] args) {  Animals a = new Animals();  a.enjoy(); // Animals enjoy!  System.out.println(a.age); // 10    Dogg d = new Dogg();  d.enjoy(); // Dog enjoy!  System.out.println(d.age); //20    Animals d1 = new Dogg();  d1.enjoy(); //Dog enjoy!  System.out.println(d1.age); // 10  Dogg s = (Dogg)d1;  System.out.println(s.age); //20    }  } |   可见，父类和子类的变量同时存在，即使同名。  子类中看到的是子类的变量，父类中看到的是父类中的变量。  互相隐藏，而同名的方法则是实实在在的覆盖（重写）。  如Animals d1 = new Dogg();  d1即是一个Animals的对象，也是一个Dogg的对象，   * 那么调用方法时，是根据对象的实际类型调用的，(看右边)   实际类型是Dogg，所以调用子类的方法。   * 是Animals时，访问的是父类的成员变量（看左边），   转型为Dogg的话，访问的就是子类的成员变量了。 |

**4.2.2 .组合与继承的区别**

**4.2.2.1．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。

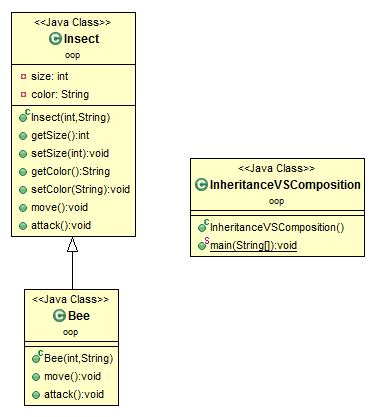
Bee（蜜蜂）是Insect（昆虫）的一种，但实现了不同于Insect（昆虫）的attack()和move方法。

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。

**4.2.2.2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。

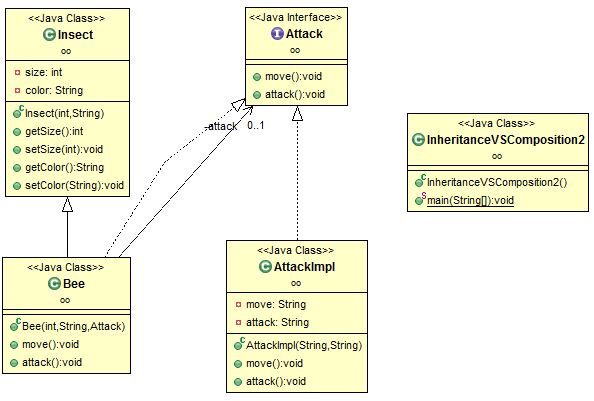
|  |
| --- |
| class AttackImpl implements Attack {  private String move;  private String attack;    public AttackImpl(String move, String attack) {  this.move = move;  this.attack = attack;  }    @Override  public void move() {  System.out.println(move);  }    @Override  public void attack() {  move();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

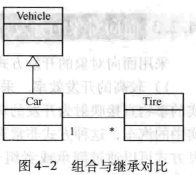
sting

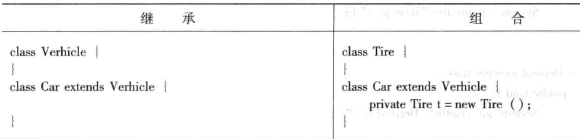
**4.2.2.3 总结**

1.组合指在新类里创建原有类的对象，重复用已有类的功能。

都允许在新类中设置子对象，组合是显示的，继承是隐式的。

组合中整体类对应继承中子类；组合中局部类对应继承中父类。





2、什么时候用继承，什么时候用组合？

* 除非存在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，才用继承机制。因继承会破坏代码可维护性，父类修改时会影响子类。
* 不要仅为了实现多态而用继承。如类间没有IS-A关系，应用组合。如策略模式。
* 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。

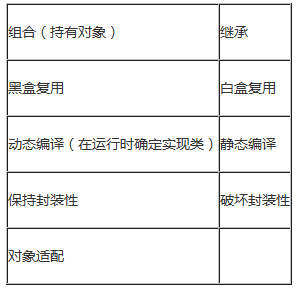
组合是运用于复杂的设计，关系在运行时才确定，即在对对象没有创建运行前，整体类是不会知道自己将持有特定接口下的哪个实现类。在扩展方面组合比继承更具有广泛性。

* 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。

**4.2.3 .多态的实现机制**

- **多态性**：允许不同子类型的对象对同一消息作出不同的响应。就是用同样的对象引用调用同样的方法但是做了不同的事情。

分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当A系统访问B系统提供的服务时，B系统有多种提供服务的方式，但一切对A系统来说都是透明的（就像电动剃须刀是A系统，它的供电系统是B系统，B系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A系统只会通过B类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。

* 两种变现方式

|  |
| --- |
| * 方法重载（overload）实现的是编译时的多态性（也称为前绑定），一个类中方法的多态性。 * 方法重写（override）实现的是运行时的多态性（也称为后绑定）。是面向对象最精髓的东西。引用变量所指向的具体实例对象的方法，即内存中正在运行的对象的方法，而不是引用变量的类型中定义的方法。 |

要实现多态：

|  |
| --- |
| 1).方法重写（子类继承父类并重写父类中已有的或抽象的方法）；  2).对象造型（用父类型引用引用子类型对象，同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。 |

**4.2.7.重载**（Overload）**和覆盖/**重写（Override）**区别**

答：

(1)都是实现多态的方式

(2)重载:编译时的多态性

重写:运行时的多态性

(3)

* 重载发生在一个类中，同名方法有不同的参数列表（参数类型、参数个数、参数顺序）.

不能通过方法访问权限、返回值类型、抛出的异常来进行重载

* 重写发生在子类与父类之间，在子类中定义某方法与其父类有相同的名称和参数及返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。基类中private方法不可重写，否则子类只是定义了一个方法。

(如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类)

重载对返回类型没有特殊的要求。

* "为什么不能根据返回类型来区分重载"?

对于重载方法，不要试图通过给予不同的返回值类型（参数列表完全相同）加以区分。因为方法在调用时可以不将返回值赋值给一个对应的变量，这样就没有特征区分。也不要试图通过在调用这样的重载方法时赋值给对应的变量加以区分，因为问题是出现在重载方法定义上，而不是调用上。总而言之，Java不允许仅仅利用返回值的不同来区分重载方法。

* 区别

1. 重载：一个类中方法间关系，水平关系；

覆盖：父类和子类间关系，垂直关系

1. 重载：参数列表不同

覆盖：参数列表相同

1. 重载：根据调用时实参表与形参表选择方法体

覆盖：根据对象(对应存储空间)类型

**4.2.8抽象类（abstractclass）与接口（interface）有什么异同**

答：

* 抽象类：

实现时，必须包含相同或更低的访问级别

不能实例化，但可创建一个对象使其指向具体子类的实例。

* 接口

可看做抽象类的变体

**相同点:**

* 抽象类和接口都不能实例化，但可定义抽象类和接口类型的引用。
* 一个类如果继承了某抽象类或实现了某接口, 都需对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。

**不同点：**

* 接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全都是抽象方法。
* 抽象类中成员可是private、默认、protected、public，而接口中的成员全都是public的。
* 抽象类中可定义成员变量(默认default：本包可见)，可在子类中重新定义及赋值。其抽象方法不能用private static synchronized native修饰.

接口中定义的成员变量实际上都是常量(默认为public static final),且必须赋值;成员方法默认都是public、abstract, 不可用static、final修饰

所以，功能需累积时，用抽象类；反之，用接口

* 有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。
* 抽象类强调所属关系，设计理念为”is-a”;

接口强调特定功能的实现，设计理念是“has-a”

* 接口用于实现较常用的功能，便于日后维护或增删方法

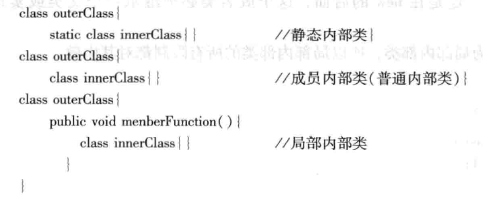
抽象类倾向于充当公共类角色，不适于日后对其中代码修改

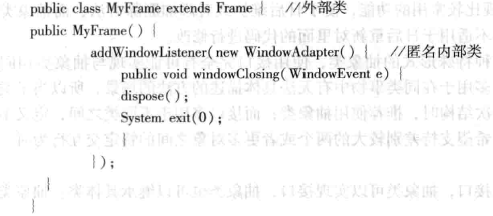
* 接口可继承接口

抽象类可实现接口，抽象类可继承具体类，抽象类也可由静态main方法。

**4.2.9内部类有哪些**

内部类可看做外部类的一个成员。





* 静态内部类：static的内部类，可不依赖于外部类实例而被实例化，不能与外部类同名，只能访问外部类中静态成员/方法(包括private).可用private,abstract等修饰
* 成员内部类(非静态内部类)：可引用外部类静态或非静态属性和方法。但与一个实例绑定，不可定义静态属性和方法。外部类实例化后才能实例化。
* 局部内部类：代码块内的类。最少使用。类似局部变量，不能被public，protected,static修饰，只能访问方法中final类型的局部变量

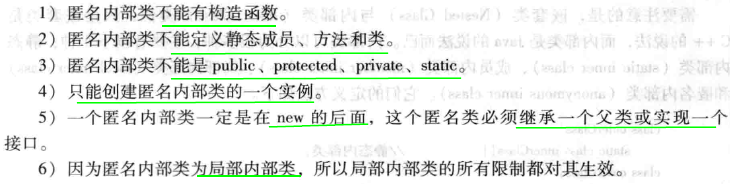
局部静态内部类:静态内部类,去掉static,移入外部类的静态方法或静态代码块中

局部内部类：成员类移入外部类的实例方法或实例初始化代码块中



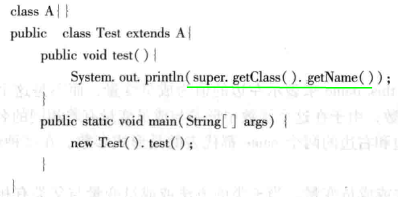
* 匿名内部类：没类名。不用class,extends,implements,没构造函数，必须继承其他类会实现其它接口。好处是使代码简洁紧凑，但易读性下降。用于GUI。

原则:

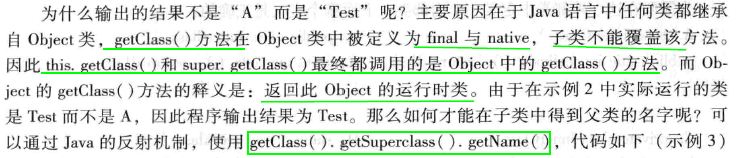


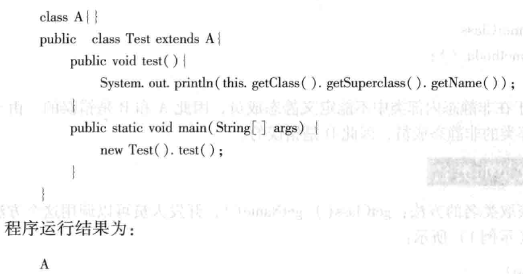
**4.2.10如何获取父类的类名**

1.



结果为:Test





**4.2.11this与super有什么区别**

**4.3关键字**

**4.3.1变量命名有哪些规则**

**4.3.2break、continue以及return有什么区别**

**4.3.3final、finally和finalize有什么区别**

**4.3.4assert有什么作用**

**4.3.5static关键字有哪些作用**

**4.3.6使用switch时有哪些注意事项**

**4.3.7volatile有什么作用**

**4.3.8instanceof有什么作用**

**4.3.9strictfp有什么作用**

**4.4基本类型与运算**

**4.4.1Java提供了哪些基本数据类型**

**4.4.2什么是不可变类**

**4.4.3值传递与引用传递有哪些区别**

**4.4.4不同数据类型的转换有哪些规则**

**4.4.5强制类型转换的注意事项有哪些**

**4.4.6运算符优先级是什么？**

**4.4.7Math类中round、ceil和floor方法的功能各是什么**

**4.4.8++i与i++有什么区别**

**4.4.9如何实现无符号数的右移操作**

**4.4.10char型变量中是否可以存储一个中文汉字**

**4.5字符串与数组**

**4.5.1字符串创建与存储的机制是什么**

**4.5.2“==”、equals和hashCode有什么区别**

**4.5.3String、StringBuffer、StringBuilder和**

**StringTokenizer有什么区别**

**4.5.4Java中数组是不是对象**

**4.5.5数组的初始化方式有哪几种**

**4.5.6length属性与length（）方法有什么区别**

**4.6异常处理**

**4.6.1finally块中的代码什么时候被执行**

**4.6.2异常处理的原理是什么**

**4.6.3运行时异常和普通异常有什么区别**

**4.7输入输出流**

**4.7.1JavaIO流的实现机制是什么**

**4.7.2管理文件和目录的类是什么**

**4.7.3JavaSocket是什么**

**4.7.4JavaNIO是什么**

**4.7.5什么是Java序列化**

**4.7.6System.out.println（）方法使用需要注意哪些问题**

**4.8Java平台与内存管理**

**4.8.1为什么说Java是平台独立性语言**

**4.8.2Java平台与其他语言平台有哪些区别**

**4.8.3JVM加载class文件的原理机制是什么**

**4.8.4什么是GC**

**4.8.5Java是否存在内存泄露问题**

**4.8.6Java中的堆和栈有什么区别**

**4.9容器**

**4.9.1JavaCollections框架是什么**

**4.9.2什么是迭代器**

**4.9.3ArrayList、Vector和LinkedList有什么区别**

**4.9.4HashMap、HashTable、TreeMap和WeakHashMap有哪些区别**

**4.9.5用自定义作为HashMap或HashTable的key需要注意哪些问题**

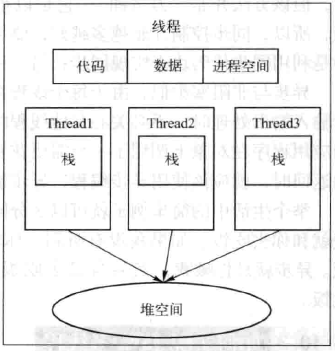
**4.9.6Collection和Collections有什么区别**

**4.10多线程**

**4.10.1什么是线程？它与进程有什么区别？为什么要使用多线程**

线程:能执行程序代码的一个执行单元。运行、就绪、挂起、结束

进程:一段正执行的程序。操作系统级别，以进程为单位。



线程间共享程序的内存空间(代码段、数据段、堆空间)及一些进程级资源(如打开的文件)，但线程有自己的栈空间。

多线程的便利:  
(1)减少程序的响应时间:耗时的线程分配到单独线程中执行。

(2)线程创建和切换开销更小。因共享代码段、数据段。数据共享效率高。

(3)多CPU或多核计算机本身就具有多线程的能力。单线程造成资源巨大浪费。多线程提高CPU利用率。

(4)多线程简化程序结构。复杂进程分多线程。

**4.10.2同步和异步有什么区别**

多线程环境中数据共享问题。当多个线程访问同一个资源，需确保该资源同一时刻只能被一个线程使用。同步机制可保证资源安全。

* 同步，需获得线程对象的锁。可保证同一时刻只有一个线程可进入临界区，且在该锁释放前，其它线程只能等待。

Synchronized实现同步(同步方法、同步代码块)，但系统开销大，有时可能死锁。所以尽量避免无谓的同步控制。

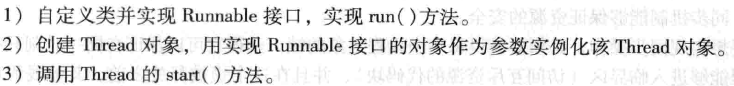
* 异步:类似非阻塞。当应用程序在对象上调用耗时方法，且不希望让程序等待方法返回，就应用异步编程，提高效率。

**4.10.3如何实现Java多线程**

1.继承Thread类(实现了runable接口的实例)，重写run()方法

调用start()后进入可运行态，运行由系统决定。

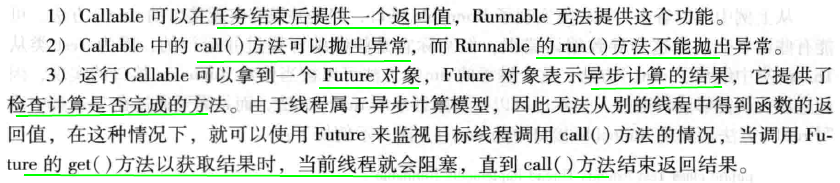
2.实现runable接口，并实现该接口的run()方法

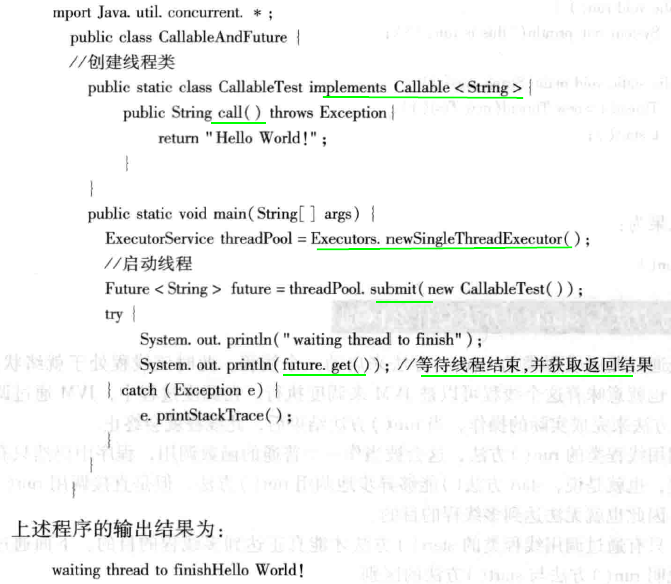


1/2都是通过Thread的对象的API来控制线程的。

3.实现Callable接口，重写call()方法

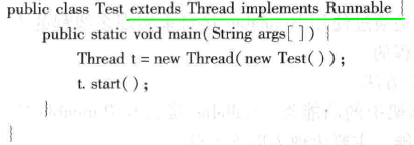
Callable接口属于Executor框架中的功能类，比Runnable更强大





一般推荐实现runnaable()接口的方式。

一个类可否同时继承Thread与实现Runnable接口?可以



**4.10.4run（）方法与start（）方法有什么区别**

**4.10.5多线程同步的实现方法有哪些**

**4.10.6sleep（）方法与wait（）方法有什么区别**

**4.10.7终止线程的方法有哪些**

**4.10.8synchronized与Lock有什么异同**

**4.10.9什么是守护线程**

**4.10.10join（）方法的作用是什么**

**4.11Java数据库操作**

**4.11.1如何通过JDBC访问数据库**

**4.11.2JDBC处理事务采用什么方法**

**4.11.3Class.forName的作用是什么**

**4.11.4Statement、PreparedStatement和CallableStatement有什么区别**

**4.11.5getString（）方法与getObject（）方法有什么区别**

**4.11.6使用JDBC时需要注意哪些问题**

**4.11.7什么是JDO**

**4.11.8JDBC与Hibernate有什么区别**

**第5章JavaWeb**

**5.1Servlet与JSP**

**5.1.1页面请求的工作流程是怎样的**

**5.1.2HTTP中GET与POST方法有什么区别**

**5.1.3什么是Servlet**

**5.1.4doPost（）方法与doGet（）方法怎么选择**

**5.1.5什么是Servlet的生命周期**

**5.1.6JSP有哪些优点**

**5.1.7JSP与Servlet有何异同**

**5.1.8如何使用JSP与Servlet实现MVC模型**

**5.1.9Servlet中forward和redirect有什么区别**

**5.1.10JSP的内置对象有哪些**

**5.1.11request对象主要有哪些方法**

**5.1.12JSP有哪些动作**

**5.1.13JSP中include指令和include动作有什么区别**

**5.1.14会话跟踪技术有哪些**

**5.1.15Web开发中如何指定字符串的编码**

**5.1.16什么是Ajax**

**5.1.17cookie和session有什么区别**

**5.2J2EE与EJB**

**5.2.1什么是J2EE**

**5.2.2J2EE中常用的术语有哪些**

**5.2.3EJB有哪些不同的类别**

**5.2.4EJB与JavaBean有什么异同**

**5.2.5EJB有哪些生命周期**

**5.2.6EJB的角色有哪几种**

**5.2.7EJB的开发流程是怎样的**

**5.2.8EJB3.0与EJB2.0有哪些不同之处**

**5.2.9EJB容器有哪些作用**

**5.2.10EJB规范规定EJB中禁止的操作有哪些**

**5.2.11Web服务器与Web应用服务器有什么区别**

**5.2.12什么是WebService**

**5.2.13SOAP与REST有什么区别**

**5.2.14什么是XML**

**5.2.15数据库连接池的工作机制是怎样的**

**5.2.16J2EE开发有哪些调优的方法**

**5.3框架**

**5.3.1什么是Struts框架**

**5.3.2Struts框架响应客户请求的工作流程是什么**

**5.3.3Struts框架的数据验证可分为几种类型**

**5.3.4FormBean的表单验证流程是什么**

**5.3.5在Struts配置文件中元素包含哪些属性和子元**

**5.3.6ActionFormBean的作用有哪些**

**5.3.7ActionForm的执行步骤有哪些**

**5.3.8forward与global-forward有什么区别**

**5.3.9Struts如何实现国际化**

**5.3.10Struts1与Struts2有哪些区别**

**5.3.11什么是IoC**

**5.3.12什么是AOP**

**5.3.13什么是Spring框架**

**5.3.14什么是Hibernate**

**5.3.15什么是Hibernate的二级缓存**

**5.3.16Hibernate中session的update（）和saveOrUpdate（）、load（）和get（）有什么区别**

**5.3.17Hibernate有哪些主键生成策略**

**5.3.18如何实现分页机制**

**5.3.19什么是SSH**

**第6章数据库原理**

**6.1SQL语言的功能有哪些**

**6.2内连接与外连接有什么区别**

**6.3什么是事务**

**6.4什么是存储过程？它与函数有什么区别与联系**

**6.5各种范式有什么区别**

**6.6什么是触发器**

**6.7什么是游标**

**6.8如果数据库日志满了，会出现什么情况**

**6.9union和unionall有什么区别**

**6.10什么是视图**

**第7章设计模式**

**7.1什么是单例模式**

**7.2什么是工厂模式**

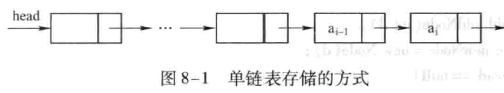
**7.3什么是适配器模式**

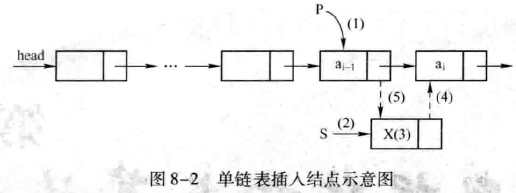
**7.4什么是观察者模式**

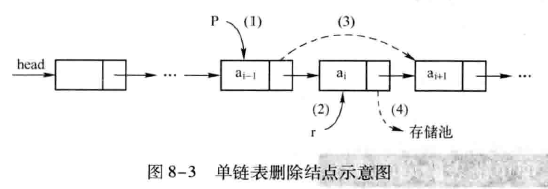
**8.数据结构与算法**

**8.1 链表**

**8.1.1 单链表的增删**

1. 





2.代码

|  |
| --- |
| package datastructure.linkedlist;  public class MyLinkedList {  Node head = null;    /\*\*  \* 向链表中插入数据  \* add node to the end  \*/  public void addNode(int d){  Node newNode = new Node(d);  if(head == null) {//空链表  head = newNode;  return;  }    Node tmp = head;  while(tmp.next != null) {//非最后一个节点  tmp = tmp.next;  }    //add node to the end  tmp.next = newNode;  }    /\*\*  \* 删除第index个节点  \* 成功返回true,失败返回false  \*/  public Boolean deleteNode(int index) {  if(index < 1 || index > length()) {//删除元素的位置不合理  return false;  }    //删除链表第一个元素  if(index == 1) {  head = head.next;  return true;  }    //其它位置  int i = 1;  Node preNode = head;  Node curNode = preNode.next;  while(curNode != null) {  if(i == index) {  preNode.next = curNode.next;  return true;  }    preNode = curNode;  curNode = curNode.next;  i++;  }  return true;  }    /\*\*  \* 返回节点长度  \*/  public int length() {  int length = 0;  Node tmp = head;  while(tmp != null) {  length++;  tmp= tmp.next;  }  return length;  }    /\*\*  \* 对链表进行排序  \* 返回排序后的头结点  \* 自己理解:冒泡法  \* 书:插入排序???  \*/  public Node orderList() {  Node nextNode = null;  int tmp=0;  Node curNode = head;  while (curNode.next != null) {//非最后一个节点  nextNode = curNode.next;  while(nextNode != null) {//【1】  if(curNode.data > nextNode.data) {  //若左边节点数据大于右边，大的换到右边  tmp = curNode.data;  curNode.data = nextNode.data;  nextNode.data = tmp;  }    //继续比较下个节点与下下个节点大小。  nextNode = nextNode.next;  }    //【2】指针指向下下个节点，依次往由，两两比较，交换。  curNode = curNode.next;  }  return head;  }    public void printList(){  Node tmp = head;  while (tmp != null) {  System.out.println(tmp.data);  tmp = tmp.next;  }  }    public static void main(String[] args) {  MyLinkedList list = new MyLinkedList();  list.addNode(5);  list.addNode(3);  list.addNode(1);  list.addNode(3);  System.out.println("list length:" + list.length());  System.out.println("before order:");  list.printList();    list.orderList();    System.out.println("after order:");  list.printList();  }  } |

* 结果：

|  |
| --- |
| list length:4  before order:  5  3  1  3  after order:  1  3  3  5 |

**8.1.2如何从链表中删除重复数据**

* 方法1：遍历链表，值存在hashtable中。遍历时，若当前访问的值在hashtable中已存在，则为重复，可删除。

优点:时间复杂度低

缺点：遍历时，需额外存储空间保存已遍历的值

|  |
| --- |
| /\*\*  \* 8.1.2 链表去重—Hashtable  \* @param head  \*/  public void deleteDuplecate1(Node head) {  Hashtable<Integer, Integer> table = new Hashtable<Integer, Integer>();  Node tmp = head;  Node pre = null;    while(tmp != null) {  if(table.containsKey(tmp.data)) {  pre.next = tmp.next;//删除重复的那个tmp  }else {  table.put(tmp.data, 1);  pre = tmp;//指针移动  }  tmp = tmp.next;  }  } |

* 方法2

双重循环遍历，外循环正常遍历链表，假设外循环当前遍历结点为芃，内循环从p开始遍历，若碰到与p所指向结点值相同，则删除这个重复结点：

优点:不需额外存储空间

缺点:时间复杂度比方法1高。

|  |
| --- |
| /\*\*  \* 8.1.2 链表去重  \* @param head  \*/  public void deleteDuplecate(Node head) {  Node p = head;  while(p.next != null) {  Node q = p;    while (q.next != null) {  if(p.data == q.next.data) {  q.next = q.next.next;//删除重复的那个q结点  } else {  q = q.next;  }  }  p = p.next;  }  } |

**8.1.3【补代码】如何找出单链表中的倒数第k个元素**

**8.1.4如何实现链表的反转(非递归方法)**

1.i m n三个节点

遍历到结点m,需要调整节点m的next指针，让它指向结点i. 但注意一旦调整指针指向，链表就断开了，因没有指针指向结点n，于是需在调整m的next之前把n保存下来。

|  |
| --- |
| /\*\*  \* 8.1.4 链表反转  \* @param head  \*/  public void ReverseTteratively(Node head) {  Node pReversedHeadNode = head;  Node pNode = head;  Node pPrev = null;  while (pNode != null) {  Node pNext = pNode.next;  if(pNext == null) {  pReversedHeadNode = pNode;  }  pNode.next = pPrev;    pPrev = pNode; //指针移动  pNode = pNext; // 指针移动  }  this.head = pReversedHeadNode;  } |

3.理解

|  |  |  |  |
| --- | --- | --- | --- |
| pPrev | pNode | pNext |  |
| (指针移动，  继续循环) | pPrev | pNode |  |

**8.1.5如何从尾到头输出单链表**

1.思路1:把链表中链接结点的指针反转过来，改变链表方向，就可从头到尾输出了，但需额外的操作。

2.思路2：从头到尾遍历链表，每经过一个结点，就放入栈中。遍历完链表后，再从栈顶开始输出结点的值，此时输出的结点顺序已经反转了。

但需维护额外的栈空间，实现较麻烦

3.思路3：更高效。递归本质上就是一个栈结构。每访问一个结点，先递归输出它后面的结点，再输出该结点自身。

|  |
| --- |
| /\*\*  \* 8.1.5 链表倒序输出  \* @param heaNode  \*/  public void printListReverse(Node heaNode) {  if(heaNode!=null) {  printListReverse(heaNode.next);  System.out.println(heaNode.data);  }  } |

**8.1.6【补代码】如何寻找单链表的中间结点**

**8.1.7【补代码】如何检测一个链表是否有环**

**8.1.8【补代码】如何在不知道头指针的情况下删除指定结点**

**8.1.9【补代码】如何判断两个链表是否相交**

**8.2栈与队列**

**8.2.1栈与队列有哪些区别**

* 又称为限定性的线性表结构:插入和删除受到更多约束和限定
* 栈:像很窄的桶，先存进去的数据只能最后取，即LIFO(后进先出)；
* 队列:像排队买东西的队列，先排先买。FIFO(先进先出)

**8.2.2【代码补】如何实现栈**

* 数组
* 链表

**8.2.3【补】如何用O（1）的时间复杂度求栈中最小元素**

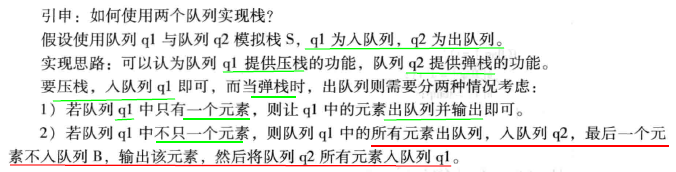
**8.2.4【补】如何实现队列**

**8.2.5如何用两个栈模拟队列操作**

|  |
| --- |
| package datastructure.queue;  import java.util.Stack;  /\*\*  \* 用两个栈模拟队列(用Stack实现)  \* 栈A(插入栈) 栈B(弹出栈) 实现队列Q  \* 1.入队列：入栈A即可  \* 2.出队列  \* (1)栈B不为空，弹出B的数据  \* (2)栈B为空:弹出A的数据—放入栈B中—再弹出栈B的数据  \* @param <E>  \*/  public class MyQueue<E> {  private Stack<E> s1 = new Stack<E>();  private Stack<E> s2 = new Stack<E>();    //入队列:A入栈  public synchronized void put(E e) {  s1.push(e);  }    //出队列  public synchronized E pop() {  if(s2.isEmpty()) {  while (!s1.isEmpty()) {  E pop = s1.pop();  s2.push(pop);  }  }  return s2.pop();  }    public static void main(String[] args) {  MyQueue<Integer> q = new MyQueue<Integer>();  q.put(1);  q.put(2);  System.out.println("队列首元素:" + q.pop());  System.out.println("队列首元素:" + q.pop());  }  } |

结果:

|  |
| --- |
| 队列首元素:1  队列首元素:2 |



**8.3【补】排序**

**8.3.1如何进行选择排序**

**8.3.2如何进行插入排序**

**8.3.3如何进行冒泡排序**

**8.3.4如何进行归并排序**

**8.3.5如何进行快速排序**

**8.3.6如何进行希尔排序**

**8.3.7如何进行堆排序**

**8.3.8各种排序算法有什么优劣**

**8.4位运算**

**8.4.1如何用移位操作实现乘法运算**

**8.4.2如何判断一个数是否为2的n次方**

* 方法1:O(logn)

|  |
| --- |
| package datastructure.\_04bitoperation;  /\*\*  \* 判断一个数是否是2的n次方  \* 2的n次方可表示为:2^0,2^1,2^2,...2^n  \* 用1做移位操作，判定移位后的值与给定数是否相等  \*/  public class Test {  public static boolean isPower(int n) {  if(n<1) {  return false;  }    int i = 1;  while (i<=n) {  if(i==n) {  return true;  }    i<<=1;  }  return false;  }  } |

* 方法2

2^0,2^1,2^2,...2^n的二进制为:1,10,100,…

可看出只有1位是1，其余为0

例如num=0001000，那么num-1=00001111，每一位都不相同，num&(num-1)=0

|  |
| --- |
| public static boolean isPower2(int n) {  if(n<1) {  return false;  }    int m = n&(n-1);  return m==0;  } |

**8.4.3如何求二进制数中1的个数**

**8.5数组**

**8.5.1如何寻找数组中的最小值与最大值**

**8.5.2如何找出数组中第二大的数**

**8.5.3如何求最大子数组之和**

**8.5.4如何找出数组中重复元素最多的数**

**8.5.5如何求数组中两两相加等于20的组合种数**

**8.5.6如何把一个数组循环右移k位**

**8.5.7如何找出数组中第k个最小的数**

**8.5.8如何找出数组中只出现次的数字**

**8.5.9如何找出数组中唯一的重复元素**

**8.5.10如何用递归方法求一个整数数组的最大元素**

**8.5.11如何求数对之差的最大值**

**8.5.12如何求绝对值最小的数**

**8.5.13如何求数组中两个元素的最小距离**

**8.5.14如何求指定数字在数组中第一次出现的位置**

**8.5.15如何对数组的两个子有序段进行合并**

**8.5.16如何计算两个有序整型数组的交集**

**8.5.17如何判断一个数组中数值是否连续相邻**

**8.5.18如何求解数组中反序对的个数**

**8.5.19如何求解最小三元组距离**

**8.6字符串**

**8.6.1如何实现字符串的反转**

**8.6.2如何判断两个字符串是否由相同的字符组成**

**8.6.3如何删除字符串中重复的字符**

**8.6.4如何统计一行字符中有多少个单词**

**8.6.5如何按要求打印数组的排列情况**

**8.6.6如何输出字符串的所有组合**

**8.7二叉树**

**8.7.1二叉树基本概念**

**8.7.2如何实现二叉排序树**

**8.7.3如何层序遍历二叉树**

**8.7.4已知先序遍历和中序遍历，如何求后序遍历**

**8.7.5如何求二叉树中结点的最大距离**

**8.8其他**

**8.8.1如何消除嵌套的括号**

**8.8.2如何不使用比较运算就可以求出两个数的最大值与最小值**

第4章Java基础知识

4.1基本概念

4.1.1Java语言有哪些优点

4.1.2Java与C/C++有什么异同

4.1.3为什么需要publicstaticvoidmain（String［］args）这个方法

4.1.4如何实现在main（）方法执行前输出“HelloWorld”

4.1.5Java程序初始化的顺序是怎样的

4.1.6Java中的作用域有哪些

4.1.7一个Java文件中是否可以定义多个类

4.1.8什么是构造函数

4.1.9为什么Java中有些接口没有任何方法

4.1.10Java中的clone方法有什么作用

4.1.11什么是反射机制

4.1.12package有什么作用

4.1.13如何实现类似于C语言中函数指针的功能

4.2面向对象技术

4.2.1面向对象与面向过程有什么区别

4.2.2面向对象有哪些特征

4.2.3面向对象的开发方式有什么优点

4.2.4什么是继承

4.2.5组合和继承有什么区别

4.2.6多态的实现机制是什么

4.2.7重载和覆盖有什么区别

第9章海量数据处理

9.1问题分析

9.2基本方法

9.3经典实例分析

9.3.1topK问题

9.3.2重复问题

9.3.3排序问题

附录

附录A软件企业Java笔试真题1

附录B软件企业Java笔试真题2

附录C软件企业Java笔试真题3

附录D求职有用网站及QQ群

一览表

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