15·常用的设计模式（概率80%）

**单例设计模式(例如：Application)**

所谓单例设计模式简单说就是无论程序如何运行，采用单例设计模式的类（Singleton类）永远只会有一个实例化对象产生。具体实现步骤如下：

(1) 将采用单例设计模式的类的构造方法私有化（采用private修饰）。

(2) 在其内部产生该类的实例化对象，并将其封装成private static类型。

(3) 定义一个静态方法返回该类的实例。

**工厂设计模式**

程序在接口和子类之间加入了一个过渡端，通过此过渡端可以动态取得实现了共同接口的子类实例化对象

**代理设计模式**

指由一个代理主题来操作真实主题，真实主题执行具体的业务操作，而代理主题负责其他相关业务的处理。比如生活中的通过代理访问网络，客户通过网络代理连接网络（具体业务），由代理服务器完成用户权限和访问限制等与上网相关的其他操作（相关业务）。

**观察者设计模式(自定义控件的时候设置点击事件,)**

所谓观察者模式，举个例子现在许多购房者都密切观察者房价的变化，当房价变化时，所有购房者都能观察到，以上的购房者属于观察者，这便是观察者模式。

java中可以借助Observable类和Observer接口轻松实现以上功能。当然此种模式的实现也不仅仅局限于采用这两个类。

观察者模式本质上就是设置监听:通过监听来观察发生变化,通过回调返回,

首先需要定义一个接口,接口里面需要定义一个方法,方法的参数就是需要返回的参数,

其次需要定义一个监听设置的方法

最后在需要监听的地方设置监听

在需要调用的的地方首先new出对象,调用监听的方法,设置内部实现类,实现方法,从而达到监听的目的.

**适配器模式(listview,gridview)**

如果一个类要实现一个具有很多抽象方法的接口，但是本身只需要实现接口中的部分方法便可以达成目的，所以此时就需要一个中间的过渡类，但此过渡类又不希望 直接使用，所以将此类定义为抽象类最为合适，再让以后的子类直接继承该抽象类便可选择性的覆写所需要的方法，而此抽象类便是适配器类。

201. 单例模式和抽象工厂各自的优点（概率35%）

单例模式：减少创建对象的次数，对内存的管理更方便，占用内存更少

抽象工厂：把公共的部分抽取出，提高了代码的复用性，也减少了代码量，便于管理。

127.知道哪几种排序方式（概率90%）

冒泡

public static int[] doMaoPao(int[] in){

int temp;

int length = in.length;

for(int i = 0;i<length; i++){

for(int j=0;j<length-1-i;j++){

if(in[j]<in[j+1]){

temp = in[j+1];

in[j+1] = in[j];

in[j] = temp;

}

System.out.print("输出j="+in[j]+"");

System.out.println("");

System.out.print("输出j+1= "+in[j+1]);

}

//System.out.print("i="+in[i]);

}

return in;

}

选择

public static int[] doPaiXv(int[] in){

for(int i=0;i<in.length;i++){

int temp = 0;

for(int j=i+1;j<in.length;j++){

if(in[i]>in[j]){

temp = in[i];

in[i] = in[j];

in[j] = temp;

}

System.out.print("输出j="+in[j]+"");

System.out.println("");

System.out.print("输出i= "+in[i]);

}

}

return in;

}

二分

public static void binary\_sort(int a[]){

int i,j,temp;

int left,right,mid;

for(i = 1; i<a.length; i++){

left = 0;

right = i-1;

temp = a[i];

while(left <= right){

mid = (left + right) /2;

if(a[mid] > temp){

right = mid - 1;

}else{

left = mid + 1;

}

}

for(j=i-1; j>right; j--){

a[j+1] = a[j];

a[right+1] = temp;

}

}

}

public static void main(String[] args) {

int a[] = {2,6,9,8,4,7,3,1,0,5};

binary\_sort(a);

for(int i=0;i<a.length;i++)

System.out.print(a[i]+ "");

}

203. 简单讲解一下递归（概率30%）

[递归算法](http://baike.baidu.com/view/1733593.htm" \t "http://blog.csdn.net/dangai0201/article/details/_blank)一般用于解决三类问题：

(1)数据的定义是按递归定义的。（Fibonacci[函数](http://baike.baidu.com/view/15061.htm" \t "http://blog.csdn.net/dangai0201/article/details/_blank)）

(2)问题解法按[递归算法](http://baike.baidu.com/view/1733593.htm" \t "http://blog.csdn.net/dangai0201/article/details/_blank)实现。

这类问题虽则本身没有明显的递归结构，但用递归求解比迭代求解更简单，如Hanoi问题。

(3)数据的结构形式是按递归定义的。

如二叉树、广义表等，由于结构本身固有的递归特性，则它们的操作可递归地描述。

递归的缺点：

[递归算法](http://baike.baidu.com/view/1733593.htm" \t "http://blog.csdn.net/dangai0201/article/details/_blank)解题相对常用的算法如普通循环等，运行效率较低。因此，应该尽量避免使用递归，除非没有更好的[算法](http://baike.baidu.com/view/7420.htm" \t "http://blog.csdn.net/dangai0201/article/details/_blank)或者某种特定情况，递归更为适合的时候。在[递归调用](http://baike.baidu.com/view/1265506.htm" \t "http://blog.csdn.net/dangai0201/article/details/_blank)的过程当中系统为每一层的返回点、局部量等开辟了栈来存储。递归次数过多容易造成[栈溢出](http://baike.baidu.com/view/2061755.htm" \t "http://blog.csdn.net/dangai0201/article/details/_blank)等。

递归典型问题：梵塔问题（汉诺塔问题）

已知有三根针分别用A, B, C表示，在A中从上到下依次放n个从小到大的盘子，现要求把所有的盘子

从A针全部移到B针，移动规则是：可以使用C临时存放盘子，每次只能移动一块盘子，而且每根针上

不能出现大盘压小盘，找出移动次数最小的方案.