**作业：**

**public** **class** PrintThread {

**public** **static** Boolean *isThreadA* = **true**;

**public** **static** Boolean *isThreadB* = **false**;

**public** **static** Boolean *isThreadC* = **false**;

/\*\*

\* notifyAll使所有原来在该对象上等待被notify的线程统统退出wait的状态，变成等待该对象上的锁，一旦该对象被解锁，他们就会去竞争。

\* notify则文明得多他只是选择一个wait状态线程进行通知，并使它获得该对象上的锁，但不惊动其他同样在等待被该对象notify的线程们，

\* 当第一个线程运行完毕以后释放对象上的锁此时如果该对象没有再次使用notify语句，则即便该对象已经空闲，其他wait状态等待的线程

\* 由于没有得到该对象的通知，继续处在wait状态，直到这个对象发出一个notify或notifyAll，它们等待的是被notify或notifyAll，而不是锁。

\* **@param** args

\*/

**public** **static** **void** main(String[] args) {

**final** PrintThread printThread = **new** PrintThread();

**new** Thread(**new** Runnable() {

**public** **void** run() {

**for** (**int** i = 0; i < 5; i++) {

**synchronized** (printThread) {

**while**(!*isThreadA*) {

**try** {

printThread.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

System.***out***.print("A");

*isThreadA* = **false**;

*isThreadB* = **true**;

*isThreadC* = **false**;

printThread.notifyAll();

}

}

}

}).start();

**new** Thread(**new** Runnable() {

**public** **void** run() {

**for** (**int** i = 0; i < 5; i++) {

**synchronized** (printThread) {

**while**(!*isThreadB*) {

**try** {

printThread.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

System.***out***.print("B");

*isThreadA* = **false**;

*isThreadB* = **false**;

*isThreadC* = **true**;

printThread.notifyAll();

}

}

}

}).start();

**new** Thread(**new** Runnable() {

**public** **void** run() {

**for** (**int** i = 0; i < 5; i++) {

**synchronized** (printThread) {

**while**(!*isThreadC*) {

**try** {

printThread.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

System.***out***.print("C");

*isThreadA* = **true**;

*isThreadB* = **false**;

*isThreadC* = **false**;

printThread.notifyAll();

}

}

}

}).start();

}

}

**第一章要点**

1、用静态工厂代替构造器

优点：具有名称便于理解代码简介，屏蔽复杂参数，便于实现单例，也实现多样初始化（返回定制类或子类）。

缺点：子类继承依赖于父类构造器

1. 遇到多个构造器是使用构建器

构建器返回的是自身，先独立保存初始化参数，然后再初始化内容

1. 用私有构造器或者枚举类强化单例模式

将实例设置为公有静态成员

通过静态工厂方法获取（灵活）

枚举类代替公有静态类型（简洁）

1. 通过私有构造器强化不可实例化的能力

子类无法访问傅磊构造器

1. 避免构建不必要的对象

使用单例模式

复用循环中的非必须对象

1. 消除过期的对象引用

强制赋值null

1. 避免使用终结方法

**第五章要点**

1、泛型使用

List与List<Object>的区别

无线通配符<?>

2、SuppressWarning

3、列表优先于数组

类型检测更加全面

4、数组与泛型转换有时候会遇到问题

泛型在运行时可以被擦除

5、优先考虑泛型

6、优先考虑泛型方法

7、通配符提升api灵活性

生产者<? extend T>

消费者<? super T>

Union.<Number>.union(...)

8、优先考虑类型安全的异构容器

favority参数类

1. **要点**

1、enum代替Int类型

从0开始，静态values方法

1. 用实例域代替序数
2. 用EnumSet代替位域
3. 用EnumMap代替序数索引
4. 用接口模拟可伸缩的枚举
5. 注解优先于命名模式
6. 坚持使用Override注解
7. 用标记接口定义类型