Effective Java

Effective Java 几个基本的原则：

1、清晰性和简洁性最为重要：模块的用户永远不应该被模块的行为所迷惑（那样就不清晰了）

2、模块要尽可能小，但又不能太小：从单个方法到包含多个包的复杂系统，都可以是一个模块。

3、代码应该被重用，而不是被拷贝。

4、模块之间的依赖性应该尽可能地降低到最小。

5、错误应该尽早被检测出来，最好在编译时刻。

第 1 条：考虑用静态工厂方法代替构造器

静态工厂方法的优势：不必每次调用它们的时候都创建一个新的对象（只初始化一份在缓存中）。如果程序经常请求创建相同的对象，并且创建对象的代价很高，则这项技术能极大地提高性能。

静态工厂方法的一些惯用名称：

valueOf：该方法返回的实例与它的参数具有相同的值

of：valueOf 的一种更为简洁的替代

getInstance：返回实例

newInstance：返回的实例与所有其它实例不同

getType：返回对象类型

newType：返回对象类型

第 2 条：遇到多个构造器参数时要考虑用构建器

重叠构造器模式：重叠构造器模式在有很多参数时，客户端代码代码会很难写，并且较难以阅读。

javaBean 模式：在遇到许多构造参数的时候，可以用替代方法，即 javaBean 模式。在这种模式下，调用一个无参构造器来创建对象，然后调用 setter 方法来设置每个必要的参数，以及相关的可选参数。javaBean 模式的缺点：由于构造过程是通过多个 set 来完成的，使得构造过程中 javaBean 可能处于不一致的状态，另外，这种模式让类变成了可变的类，因此可能是线程不安全的。

Build 模式：既能够保证像重叠构造器那样的安全性，又能够保证像 javaBeans 模式那么好的额可读性。

第 5 条：避免创建不必要的对象

当你应该重用现有对象的时候，请不要创建新的对象。

当你应该创建新的对象时，请不要重用现有的对象。

要优先使用基本类型而不是装箱基本类型，要注意无意思的自动装箱。

第 6 条：避免创建不必要的对象

只要类是自己管理内存，程序员就应该警惕内存泄露问题。一旦元素被释放掉，该元素中所

包含的任何对象应该被清空（比如设置为 null）。

第 16 条：复合优于继承

只有当子类和超类之间确实存在子类型关系时，使用继承才恰当。

第 18 条：接口优于抽象类

第 19 条：接口只用于定义类型

有一种接口被称为常量接口（constant interface），常量接口模式是对接口的不良使用，可以用枚举类型。简而言之，接口只应该用来定义类型，而不应该被用来导出常量。

第 20 条：类层次由于标签类

第 22 条：优先考虑静态成员类

如果一个嵌套类需要在某个方法之外仍然可见的，或者它太长了，不适合放在放在方法内部，就应该使用成员类；如果成员类的每个实例都需要一个指向其外围实例的引用，就要把成员类做成非静态的；否则就要做成静态的。假设这个嵌套类属于一个方法的内部，如果你需要在一个地方创建实例，并且已经有了一个预制的类型可以说明这个类的特征，就要把它做成匿名类；否则做成局部类。

问题：

有三个线程 A、B、C(线程名称或 id)，请你用今天学到的知识，循环打印 10 次 ABCABB...

提供四种方法：1、SingleThreadExecutor;2、基于 join 方法；3、信号量；4、locker 锁

方法一：使用 SingleThreadExecutor，刚方法表示线程数为 1 的 FixedThreadPool。向 SingleThreadExecutor 提交的多个任务，会按照它们的提交顺序，并且在下一个任务完成之前完成，因为所有的任务都使用相同的线程。

实现代码：

```
package cn.rbac.thread;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
public class Demo implements Runnable {
    private String threadName;
    public Demo() {
    }
    public Demo(String threadName) {
        this.threadName = threadName;
    }
    @Override
    public void run() {
        try {
```

```
            TimeUnit.SECONDS.sleep(1);
            Thread.currentThread().setName(threadName);
            System.out.print(Thread.currentThread().getName());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
        Demo demoA = new Demo("A");
        Demo demoB = new Demo("B");
        Demo demoC = new Demo("C");
        ExecutorService exec = Executors.newSingleThreadExecutor();
        for(int i=0;i<10;i++){
            exec.execute(demoA);
            exec.execute(demoB);
            exec.execute(demoC);
        }
        exec.shutdown();
    }
}
```

方法二：使用 join 方法，思路是：步骤一：在单次循环 ABC，为保证顺序，线程 C 的 run 执行插入 B.join()，线程 B 的 run 执行插入 A.join()；步骤二：为保证每次都是 ABC 为一个单位依次循环，需要在主线程中加入 A.join(),B.join(),C.join()，然后执行循环操作。

代码如下：

```
package cn.rbac.thread;
public class DemoABC2 {
    static class ThreadC extends Thread {
        private String name;
        private ThreadB threadB;

        public ThreadC(String name, ThreadB threadB) {
            this.name = name;
            this.threadB = threadB;
        }

        public void run() {
            try {
                threadB.join();
                System.out.print(name);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
static class ThreadB extends Thread {
    private String name;
    private ThreadA threadA;

    public ThreadB(String name, ThreadA threadA) {
        this.name = name;
        this.threadA = threadA;
    }

    public void run() {
        try {
            threadA.join();
            System.out.print(name);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }
}

static class ThreadA extends Thread {
    private String name;

    public ThreadA(String name) {
        this.name = name;
    }

    public void run() {
        System.out.print(name);
    }
}

public static void main(String[] args) {
    for (int i = 0; i < 10; i++) {
        ThreadA threada = new ThreadA("A");
        ThreadB threadb = new ThreadB("B", threada);
        ThreadC threadc = new ThreadC("C", threadb);
        threada.start();
        threadb.start();
        threadc.start();
        try {
            threada.join();
            threadb.join();
```

```
                threadc.join();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
    }
}
```

方法三：使用 lock

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import javax.xml.stream.events.StartDocument;
public class ABC2 {
    private static Lock lock = new ReentrantLock();
    private static int count = 0;
    private static Condition A = lock.newCondition();
    private static Condition B = lock.newCondition();
    private static Condition C = lock.newCondition();
    static class ThreadA extends Thread {
        @Override
        public void run() {
            lock.lock();
            try {
                for (int i = 0; i < 10; i++) {
                    while (count % 3 != 0)
                        A.await(); //如果不满足while条件，将本线程挂起
                    System.out.print("A");
                    count++;
                    B.signal(); // A线程执行后，唤醒下一个线程B
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

    }

    static class ThreadB extends Thread {

        @Override
        public void run() {
```

```
            lock.lock();
            try {
                for (int i = 0; i < 10; i++) {
                    while (count % 3 != 1)
                        B.await();//如果不满足while条件， 将本线程挂起
                    System.out.print("B");
                    count++;
                    C.signal();// B线程执行后，唤醒下一个线程C
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

    }

    static class ThreadC extends Thread {

        @Override
        public void run() {
            lock.lock();
            try {
                for (int i = 0; i < 10; i++) {
                    while (count % 3 != 2)
                        C.await();//如果不满足while条件， 将本线程挂起
                    System.out.println("C");
                    count++;
                    A.signal();// C线程执行后，唤醒下一个线程A
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }

    }

    public static void main(String[] args) throws InterruptedException {
        ThreadA threadA =new ThreadA();
        ThreadB threadB=new ThreadB();
        ThreadC threadC = new ThreadC();
```

```
        threadA.start();
        threadB.start();
        threadC.start();
        threadC.join();//让C线程执行完后在输出cout值否则可能cout在ABC线程都未完成时就输出结果。
        System.out.println(count);
    }
}
```

方法四：使用信号量

```
import java.util.concurrent.Semaphore;
public class ABC3 {
    private static Semaphore A = new Semaphore(1);
    private static Semaphore B = new Semaphore(1);
    private static Semaphore C = new Semaphore(1);

    static class ThreadA extends Thread {

        @Override
        public void run() {
            try {
                for (int i = 0; i < 10; i++) {
                    A.acquire();
                    System.out.print("A");
                    B.release();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

    static class ThreadB extends Thread {
        @Override
        public void run() {
            try {
                for (int i = 0; i < 10; i++) {
                    B.acquire();
                    System.out.print("B");
                    C.release();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
    }

    static class ThreadC extends Thread {

        @Override
        public void run() {
            try {
                for (int i = 0; i < 10; i++) {
                    C.acquire();
                    System.out.println("C");
                    A.release();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

    public static void main(String[] args) throws InterruptedException {
        B.acquire();
        C.acquire(); // 开始只有A可以获取，BC都不可以获取，保证了A最先执行
        new ThreadA().start();
        new ThreadB().start();
        new ThreadC().start();
    }
}
```