SONAR blocker & critical规则分析

BLOCKER问题

".equals()" should not be used to test the values of "Atomic" classes

“.equals()”方法不应该被用来测试“原子”类的值。

atomicinteger，atomiclong继续了NUMBER，但他们与integer和long不同，必须被区别对待。atomicinteger是被设计为支持无需锁、线程安全编程的简单变量。所以，一个atomicinteger将永远只能是equals于其本身。相反，你应该用get()去作比较。

这适用于所有的原子类：atomicinteger，atomiclong,atomicboolean。

AtomicInteger, and AtomicLong extend Number, but they're distinct from Integer and Long and should be handled differently. AtomicInteger and AtomicLong are designed to support lock-free, thread-safe programming on single variables. As such, an AtomicInteger will only ever be "equal" to itself. Instead, you should .get() the value and make comparisons on it.

This applies to all the atomic, seeming-primitive wrapper classes: AtomicInteger, AtomicLong, and AtomicBoolean.

Noncompliant Code Example(错误的用法)

```
AtomicInteger aInt1 = new AtomicInteger(0);
AtomicInteger aInt2 = new AtomicInteger(0);

if (aInt1.equals(aInt2)) { ... }  // Noncompliant
```

Compliant Solution（正确的解决方案）

```
AtomicInteger aInt1 = new AtomicInteger(0);
AtomicInteger aInt2 = new AtomicInteger(0);

if (aInt1.get() == aInt2.get()) { ... }
```

"Double.longBitsToDouble" should not be used for "int"

Double.longBitsToDouble不能用于int

Double.longBitsToDouble的参数要求是64位的long参数。传给它一个比较小的数值，比方说int，会引起预期之外的结果。因为类型的转换会有问题，就好比让一个小孩带大人的手套...

Double.longBitsToDouble expects a 64-bit, long argument. Pass it a smaller value, such as an int and the mathematical conversion into a double simply won't work as anticipated because the layout of the bits will be interpreted incorrectly, as if a child were trying to use an adult's gloves.

Noncompliant Code Example

```
int i = 42;
double d = Double.longBitsToDouble(i);  // Noncompliant
```

"equals(Object obj)" and "hashCode()" should be overridden in pairs

equals方法和hashcode方法必须被同时重写。

根据java语言规范，equals()和hashcode()有如下关联：

如果根据equals方法得出两个对象是相等的结论，那么调用hashCode方法得到的结果必须是相等的。

所以，这两个方法必须同是继承，或重写。

According to the Java Language Specification, there is a contract between equals(Object) and hashCode():

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results.

However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

In order to comply with this contract, those methods should be either both inherited, or both overridden.

Noncompliant Code Example

```
class MyClass {    // Noncompliant - should also override "hashCode()"

  @Override
  public boolean equals(Object obj) {
    /* ... */
  }

}
```

Compliant Solution

```
class MyClass {    // Compliant

  @Override
  public boolean equals(Object obj) {
    /* ... */
```

```
  }

  @Override
  public int hashCode() {
    /* ... */
  }

}
```

"BigDecimal(double)" should not be used

“BigDecimal（double）” 不应该使用

由于浮点精度问题，你不可能从BigDecimal()得到你期望的值。

以下文字摘自javadocs：

有人可能会认为从BigDecimal（0.1）new出来的数值正好等于0.1，但它实际上是等于0.1000000000000000055511151231257827021181583404541015625。这是因为0.1不能被精确地表示为一个双精度浮点数。因此，看起来你传入的参数是0.1，但实际并不是。

所以，你应该使用bigdecimal.valueof，它会消除浮点数带来的误差。

Because of floating point imprecision, you're unlikely to get the value you expect from the BigDecimal(double) constructor.

From the JavaDocs:

The results of this constructor can be somewhat unpredictable. One might assume that writing new BigDecimal(0.1) in Java creates a BigDecimal which is exactly equal to 0.1 (an unscaled value of 1, with a scale of 1), but it is actually equal to 0.1000000000000000055511151231257827021181583404541015625. This is because 0.1 cannot be represented exactly as a double (or, for that matter, as a binary fraction of any finite length). Thus, the value that is being passed in to the constructor is not exactly equal to 0.1, appearances notwithstanding.

Instead, you should use BigDecimal.valueOf, which uses a string under the covers to eliminate floating point rounding errors.

Noncompliant Code Example

```
double d = 1.1;

BigDecimal bd1 = new BigDecimal(d); // Noncompliant; see comment above
BigDecimal bd2 = new BigDecimal(1.1); // Noncompliant; same result
```

Compliant Solution

```
double d = 1.1;

BigDecimal bd1 = BigDecimal.valueOf(d);
BigDecimal bd2 = BigDecimal.valueOf(1.1);
```

"Cloneables" should implement "clone"

继承cloneable接口应该实现clone方法

仅仅继承了cloneable接口而没有重写clone方法并不会让这个类一定cloneable，如果这个cloneable的接口没有包括clone方法，将使用默认的JVM的克隆，复制原始值和从源到目标对象的引用。任何克隆实例都将有可能与源实例共享成员。

Simply implementing Cloneable without also overriding Object.clone() does not necessarily make the class cloneable. While the Cloneable interface does not include a clone method, it is required by convention, and ensures true cloneability. Otherwise the default JVM clone will be used, which copies primitive values and object references from the source to the target. I.e. without overriding clone, any cloned instances will potentially share members with the source instance.

Removing the Cloneable implementation and providing a good copy constructor is another viable (some say preferable) way of allowing a class to be copied.

Noncompliant Code Example

```
class Team implements Cloneable {  // Noncompliant
  private Person coach;
  private List<Person> players;
  public void addPlayer(Person p) {...}
  public Person getCoach() {...}
}
```

Compliant Solution

```
class Team implements Cloneable {
  private Person coach;
  private List<Person> players;
  public void addPlayer(Person p) { ... }
  public Person getCoach() { ... }

  @Override
  public Object clone() {
    Team clone = (Team) super.clone();
    //...
  }
}
```

"equals(Object obj)" should be overridden along with the "compareTo(T obj)" method

equals() 应该和compareTo()一起被重写

根据JAVA文档：

我们强烈建议（但并不是强制要求），如果x.equals(y)成立，那么 x.compareTo(y) 也应该成立。即(x.compareTo(y)==0) ==

(x.equals(y))，如果违反了这条规则，会出现奇怪的和不可预知的失败。例如，在java 5中，remove()方法依靠compareto()，但自从java 6它依赖于equals()。

According to the Java Comparable.compareTo(T o) documentation:

It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

If this rule is violated, weird and unpredictable failures can occur. For example, in Java 5 the PriorityQueue.remove() method relied on compareTo(), but since Java 6 it relies on equals().

Noncompliant Code Example

```
public class Foo implements Comparable<Foo> {
  @Override
  public int compareTo(Foo foo) { /* ... */ }      // Noncompliant as the equals(Object obj) method is not overridden
}
```

Compliant Solution

```
public class Foo implements Comparable<Foo> {
  @Override
  public int compareTo(Foo foo) { /* ... */ }      // Compliant

  @Override
  public boolean equals(Object obj) { /* ... */ }
}
```

"for" loop incrementers should modify the variable being tested in the loop's stop condition

for循环中，自增的变量应该与循环终止条件中的变量是同一个

一个循环终止条件中的变量与自增的变量不是同一个变量的时候，往往都是有问题。即使不是错误，将来也很容易引起后来者的误解，所以最好避免。

It is almost always an error when a for loop's stop condition and incrementer don't act on the same variable. Even when it is not, it could confuse future maintainers of the code, and should be avoided.

Noncompliant Code Example

```
for (i = 0; i < 10; j++) {  // Noncompliant
  // ...
}
```

Compliant Solution

```
for (i = 0; i < 10; i++) {
  // ...
}
```

"Iterator.hasNext()" should not call "Iterator.next()"

iterator.hasnext()不应该调用iterator.next()

调用iterator.hasnext()不应该有任何的副作用，也不应改变迭代器的状态。iterator.next()会使迭代器前进一步。所以说在hasnext里面调用next会破坏hasnext()的初衷，并将导致生产意外行为。

Calling Iterator.hasNext() is not supposed to have any side effects, and therefore should not change the state of the iterator. Iterator.next() advances the iterator by one item. So calling it inside Iterator.hasNext(), breaks the hasNext() contract, and will lead to unexpected behavior in production.

Noncompliant Code Example

```
public class FibonacciIterator implements Iterator<Integer>{
...
@Override
public boolean hasNext() {
  if(next() != null) {
    return true;
  }
  return false;
}
...
}
```

"Lock" objects should not be "synchronized"

“锁定”对象不应该是“同步”的

java.util.concurrent.locks.lock提供比同步块更强大和灵活的锁定操作。因此，在一个锁上的同步扔掉了对象的灵活性，这非常的愚蠢。相反，这样的对象如果需要锁定，请使用trylock()和unlock()。

java.util.concurrent.locks.Lock offers far more powerful and flexible locking operations than are available with synchronized blocks. So synchronizing on a Lock throws away the power of the object, and is just silly. Instead, such objects should be locked and unlocked using tryLock() and unlock().

Noncompliant Code Example

```
Lock lock = new MyLockImpl();
synchronized(lock) {  // Noncompliant
  //...
}
```

Compliant Solution

```
Lock lock = new MyLockImpl();
lock.tryLock();
//...
```

"Object.wait(...)" should never be called on objects that implement "java.util.concurrent.locks.Condition"

object.wait()不应该被实现了"java.util.concurrent.locks.Condition"的对象调用

实现条件接口的目的就是为了获得更灵活的await方法，所以一个实现了条件接口的对象调用wait方法是非常愚蠢和逻辑混乱的...

From the Java API documentation:

Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods.

The purpose of implementing the Condition interface is to gain access to its more nuanced await methods. Therefore, calling the method Object.wait(...) on a class implementing the Condition interface is silly and confusing.

Noncompliant Code Example

```
final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();
...
notFull.wait();
```

Compliant Solution

```
final Lock lock = new ReentrantLock();
final Condition notFull  = lock.newCondition();
...
notFull.await();
```

"return" statements should not occur in "finally" blocks

return语句不能出现在finally块中

从一个finally块中return会阻碍程序 捕获从try或catch块中抛出的未处理的异常。

Returning from a finally block suppresses the propagation of any unhandled Throwable which was thrown in the try or catch block.

Noncompliant Code Example

```
public static void main(String[] args) {
  try {
    doSomethingWhichThrowsException();
    System.out.println("OK");   // incorrect "OK" message is printed
  } catch (RuntimeException e) {
    System.out.println("ERROR");  // this message is not shown
  }
}

public static void doSomethingWhichThrowsException() {
  try {
    throw new RuntimeException();
  } finally {
    /* ... */
    return;      // Noncompliant - prevents the RuntimeException from being propagated
  }
}
```

Compliant Solution

```
public static void main(String[] args) {
  try {
    doSomethingWhichThrowsException();
    System.out.println("OK");
  } catch (RuntimeException e) {
    System.out.println("ERROR");  // "ERROR" is printed as expected
  }
}

public static void doSomethingWhichThrowsException() {
  try {
    throw new RuntimeException();
```

```
  } finally {
    /* ... */
  }
}
```

"runFinalizersOnExit" should not be called

"runFinalizersOnExit"不应该被调用

如果调用这个方法的时候有其他线程同时操作这些活的对象，这种行为会导致不稳定或死锁。

如果你真的想在虚拟机开始关机的时候执行某些操作，你应该附上一个关机钩。

Running finalizers on JVM exit is disabled by default. It can be enabled with System.runFinalizersOnExit and Runtime.runFinalizersOnExit, but both methods are deprecated because they are are inherently unsafe.

According to the Oracle Javadoc:

It may result in finalizers being called on live objects while other threads are concurrently manipulating those objects, resulting in erratic behavior or deadlock.

If you really want to be execute something when the virtual machine begins its shutdown sequence, you should attach a shutdown hook.

Noncompliant Code Example

```
public static void main(String [] args) {
  ...
  System.runFinalizersOnExit(true);  // Noncompliant
  ...
}

protected void finalize(){
  doSomething();
}
```

Compliant Solution

```
public static void main(String [] args) {
  Runtime.addShutdownHook(new Runnable() {
    public void run(){
      doSomething();
    }
  });
```

```
//...
```

"ScheduledThreadPoolExecutor" should not have 0 core threads

计划线程池执行器不应该有0个核心线程

计划线程池执行器的线程池大小是根据核心线程数来确定的，所以，把核心线程数设为0意味着这个线程池执行器将没有线程，也无法运行；

java.util.concurrent.ScheduledThreadPoolExecutor's pool is sized with corePoolSize, so setting corePoolSize to zero means the executor will have no threads and run nothing.

This rule detects instances where corePoolSize is set to zero, via either its setter or the object constructor.

Noncompliant Code Example

```
public void do(){

  ScheduledThreadPoolExecutor stpe1 = new ScheduledThreadPoolExecutor(0); // Noncompliant

  ScheduledThreadPoolExecutor stpe2 = new ScheduledThreadPoolExecutor(POOL_SIZE);
  stpe2.setCorePoolSize(0);  // Noncompliant
```

"toString()" and "clone()" methods should not return null

tostring() 和 clone()方法不应该返回空

调用一个对象的tostring或是clone方法总是应该返回一个string或是an object，返回一个空值违反了这个方法的隐性约定。

Calling toString() or clone() on an object should always return a string or an object. Returning null instead contravenes the method's implicit contract.

Noncompliant Code Example

```
public override string ToString () {
  if (this.collection.Count == 0) {
    return null; // Noncompliant
  } else {
    // ...
```

Compliant Solution

```
public override string ToString () {
  if (this.collection.Count == 0) {
    return "";
  } else {
    // ...
```

"wait(...)" should be used instead of "Thread.sleep(...)" when a lock is held

当线程持有锁的时候，要用wait方法，不要用sleep方法。

如果当线程持有锁的时候调用sleep方法，可能会导致性能和可扩展性的问题，甚至死锁。因为持有锁的线程的执行被冻结。这个时候应该调用wait方法，暂时释放锁并允许其他线程运行。

If Thread.sleep(...) is called when the current thread holds a lock, it could lead to performance, and scalability issues, or even worse to deadlocks because the execution of the thread holding the lock is frozen. It's better to call wait(...) on the monitor object to temporarily release the lock and allow other threads to run.

Noncompliant Code Example

```
public void doSomething(){
  synchronized(monitor) {
    while(notReady()){
      Thread.sleep(200);
    }
    process();
  }
  ...
}
```

Compliant Solution

```
public void doSomething(){
  synchronized(monitor) {
    while(notReady()){
      monitor.wait(200);
    }
    process();
  }
  ...
}
```

"wait(...)", "notify()" and "notifyAll()" methods should only be called when a lock is obviously held on an object

wait() notify() notifyall()方法只有当对象显式持有锁的时候才可以。

根据协议，wait() notify()
notifyall()方法只有当对象显式持有锁的时候才可以。否则，会引起“非法的监视器状态”异常。这条规则要求上述三个方法只能在同步的块或方法中调用。

By contract, the method Object.wait(...), Object.notify() and Object.notifyAll() should be called by a thread that is the owner of the object's monitor. If this is not the case an IllegalMonitorStateException exception is thrown. This rule reinforces this constraint by making it mandatory to call one of these methods only inside a synchronized method or statement.

Noncompliant Code Example

```
private void removeElement() {
  while (!suitableCondition()){
    obj.wait();
  }
  ... // Perform removal
}
```

or

```
private void removeElement() {
  while (!suitableCondition()){
    wait();
  }
  ... // Perform removal
}
```

Compliant Solution

```
private void removeElement() {
  synchronized(obj) {
    while (!suitableCondition()){
      obj.wait();
    }
    ... // Perform removal
  }
}
```

or

```
private synchronized void removeElement() {
  while (!suitableCondition()){
    wait();
  }
```

```
  ... // Perform removal
}
```

A "for" loop update clause should move the counter in the right direction

for循环应该在正确方向上移动计数器

一个在错误方向上移动计数器的循环不是一个无限循环。由于数据类型必定会溢出（就像int型的负数一直减1最终会变成正数...），循环将最终达到停止条件，但在这样做时，它会运行很多次，时间比预期的更多，而且可能会导致意外的行为。

A for loop with a counter that moves in the wrong direction is not an infinite loop. Because of wraparound, the loop will eventually reach its stop condition, but in doing so, it will run many, many more times than anticipated, potentially causing unexpected behavior.

Noncompliant Code Example

```
public void doSomething(String [] strings) {
  for (int i = 0; i < strings.length; i--) { // Noncompliant;
    String string = strings[i];  // ArrayIndexOutOfBoundsException when i reaches -1
    //...
  }
```

Compliant Solution

```
public void doSomething(String [] strings) {
  for (int i = 0; i < strings.length; i++) {
    String string = strings[i];
    //...
  }
```

Collections should not be passed as arguments to their own methods

集合不应该作为参数传递给它们自己的方法

把集合本身做为参数传给自己的方法往往是一个错误----应该传其它的参数----或者只是荒谬的代码。

此外，因为一些方法的参数要求在执行过程中不能被修改的，所以把本身做为参数传给自己可能导致在未定义的行为。

Passing a collection as an argument to the collection's own method is either an error - some other argument was intended - or simply nonsensical code.

Further, because some methods require that the argument remain unmodified during the execution, passing a collection to itself can result in undefined behavior.

Noncompliant Code Example

```
List <Object> objs = new ArrayList<Object>();
 objs.add("Hello");

 objs.add(objs); // Noncompliant; StackOverflowException if objs.hashCode() called
 objs.addAll(objs); // Noncompliant; behavior undefined
 objs.containsAll(objs); // Noncompliant; always true
 objs.removeAll(objs); // Noncompliant; confusing. Use clear() instead
 objs.retainAll(objs); // Noncompliant; NOOP
```

Exit methods should not be called

不应该调用退出方法

Calling System.exit(int status) or Rutime.getRuntime().exit(int status)会调用shutdown钩子并关闭整个java虚拟机。调用Runtime.getRuntime().halt(int)会立即停机，并不会调用shutdown钩子，也不会finalization。

这些方法应小心使用，只有当目的是阻止整个java程序时才可以用。例如，在J2EE容器中运行的应用都不应该调用exit方法。

Calling System.exit(int status) or Rutime.getRuntime().exit(int status) calls the shutdown hooks and shuts downs the entire Java virtual machine. Calling Runtime.getRuntime().halt(int) does an immediate shutdown, without calling the shutdown hooks, and skipping finalization.

Each of these methods should be used with extreme care, and only when the intent is to stop the whole Java process. For instance, none of them should be called from applications running in a J2EE container.

Noncompliant Code Example

```
System.exit(0);
Runtime.getRuntime().exit(0);
Runtime.getRuntime().halt(0);
```

Methods "wait(...)", "notify()" and "notifyAll()" should never be called on Thread instances

wait notify notifyall方法不应该被线程的实例调用

有两个原因促使我们不要用线程的实例去调用这三个方法：

1、这样做会引起混乱，比方说一个线程调用了wait方法，这样做的真正意图是什么？是要终止这个线程的执行，还是要等待对象监视器去捕获？

2、实际中，JVM正是通过这些方法去改变线程的状态（阻塞、等待...），所以手动的调用这些方法会扰乱JVM的行为。

On a Thread instance, the methods wait(...), notify() and notifyAll() are available only because all classes in Java extend Object and

therefore automatically inherit the methods. But there are two very good reasons to not call these methods on a Thread instance:

Doing so is really confusing. What is really expected when calling, for instance, the wait(...) method on a Thread? That the execution of the Thread is suspended, or that acquisition of the object monitor is waited for?

Internally, the JVM relies on these methods to change the state of the Thread (BLOCKED, WAITING, ...), so calling them will corrupt the behavior of the JVM.

Noncompliant Code Example

```
Thread myThread = new Thread(new RunnableJob());
...
myThread.wait(2000);
```

Non-serializable classes should not be written

非序列化的类不能写

如果将一个非序列化的类写入文件中，什么也不会写入，而且试图序列化这样的类会引发异常。只有实现或承继序列化的类可以。

Nothing in a non-serializable class will be written out to file, and attempting to serialize such a class will result in an exception being thrown. Only a class that implements Serializable or one that extends such a class can successfully be serialized (or de-serialized).

Noncompliant Code Example

```
public class Vegetable {  // neither implements Serializable nor extends a class that does
  //...
}

public class Menu {
  public void meal() throws IOException {
    Vegetable veg;
    //...
    FileOutputStream fout = new FileOutputStream(veg.getName());
    ObjectOutputStream oos = new ObjectOutputStream(fout);
    oos.writeObject(veg);  // Noncompliant. Nothing will be written
  }
}
```

Compliant Solution

```
public class Vegetable implements Serializable {  // can now be serialized
  //...
```

```
}

public class Menu {
  public void meal() throws IOException {
    Vegetable veg;
    //...
    FileOutputStream fout = new FileOutputStream(veg.getName());
    ObjectOutputStream oos = new ObjectOutputStream(fout);
    oos.writeObject(veg);
  }
}
```

super.finalize() should be called at the end of Object.finalize() implementations

super.finalize()应该在object.finalize接口的最后被调用

重写object.finalize方法务必要慎重的处理系统资源。

所以这里强烈建议在object.finalize()方法的最后调用 super.finalize()去处理父类中的系统资源。

Overriding the Object.finalize() method must be done with caution to dispose some system resources.

Calling the super.finalize() at the end of this method implementation is highly recommended in case parent implementations must also dispose some system resources.

Noncompliant Code Example

```
protected void finalize() {   // Noncompliant; no call to super.finalize();
  releaseSomeResources();
}

protected void finalize() {
  super.finalize();  // Noncompliant; this call should come last
  releaseSomeResources();
}
```

Compliant Solution

```
protected void finalize() {
  releaseSomeResources();
  super.finalize();
```

}

Synchronization should not be based on Strings or boxed primitives

同步锁不应该加在string或是装箱的原语（如Integer）

不可复用的对象不应该加同步锁。因为，这将引起不相关的线程之前的死锁。一个极端的例子就是在Boolean对象上加锁，它一共就两个取值，非常容易阻塞其它线程...

Objects which are pooled and potentially reused should not be used for synchronization. If they are, it can cause unrelated threads to deadlock with unhelpful stacktraces. Specifically, String literals, and boxed primitives such as Integers should not be used as lock objects because they are pooled and reused. The story is even worse for Boolean objects, because there are only two instances of Boolean, Boolean.TRUE and Boolean.FALSE and every class that uses a Boolean will be referring to one of the two.

Noncompliant Code Example

```
private static final Boolean bLock = Boolean.FALSE;
private static final Integer iLock = Integer.valueOf(0);
private static final String sLock = "LOCK";

public void doSomething() {

  synchronized(bLock) {  // Noncompliant
    // ...
  }
  synchronized(iLock) {  // Noncompliant
    // ...
  }
  synchronized(sLock) {  // Noncompliant
    // ...
  }
```

Compliant Solution

```
private static final Object lock1 = new Object();
private static final Object lock2 = new Object();
private static final Object lock3 = new Object();

public void doSomething() {

  synchronized(lock1) {
```

```
  // ...
}
synchronized(lock2) {
  // ...
}
synchronized(lock3) {
  // ...
}
```

CRITICAL问题

"ConcurrentLinkedQueue.size()" should not be used

一般而言，size()方法的执行耗时是固定的，但是，执行ConcurrentLinkedQueue.size()方法需要的时间是O(n)的，即与队列中元素的个数成正比的。当队列很大时，这可能是一个昂贵的操作。此外，结果可能是不准确的，例如在执行过程中的队列被修改。

顺便说一下，如果size()仅用于检查集合是否是空的，那么isempty()是应该使用方法。

For most collections the size() method requires constant time, but the time required to execute ConcurrentLinkedQueue.size() is directly proportional to the number of elements in the queue. When the queue is large, this could therefore be an expensive operation. Further, the results may be inaccurate if the queue is modified during execution.

By the way, if the size() is used only to check that the collection is empty, then the isEmpty() method should be used.

Noncompliant Code Example

```
ConcurrentLinkedQueue queue = new ConcurrentLinkedQueue();
//...
log.info("Queue contains " + queue.size() + " elements");
```

"hashCode" and "toString" should not be called on array instances

虽然hashCode和toString方法可以用于数组实例，但它们基本上是无用的。hashCode返回数组的“身份的哈希值”，toString也返回几乎相同的值。这两种方法的返回值都不是数组的内容。相反，你应该将数组传递给相关的静态数组方法。

While hashCode and toString are available on arrays, they are largely useless. hashCode returns the array's "identity hash code", and toString returns nearly the same value. Neither method's output actually reflects the array's contents. Instead, you should pass the array to the relevant static Arrays method.

Noncompliant Code Example

```
public static void main( String[] args )
{
   String argStr = args.toString(); // Noncompliant
```

```
    int argHash = args.hashCode(); // Noncompliant
```

Compliant Solution

```
public static void main( String[] args )
{
    String argStr = Arrays.toString(args);
    int argHash = Arrays.hashCode(args);
```

"object == null" should be used instead of "object.equals(null)"

没有经验的开发者可能会觉得Object.equals(Object obj)方法可以正确处理左手边本身为空的情况，但事实并非如此...

Inexperienced Java developers might expect the Object.equals(Object obj) method to correctly handle the case where the left hand side is null, but that is not the case.

Noncompliant Code Example

```
if (variable.equals(null)) { /* ... */ }   // Noncompliant - "variable" is really null, a NullPointerException is thrown
```

Compliant Solution

```
if (variable == null) { /* ... */ }        // Compliant
```

"public static" fields should be constant

没有什么好的理由来声明一个“公开”和“静态”的域，而不声明它是“终态”的。大多数的时间这个域是用来几个对象之间分享状态。但如果不加final，任何对象可以任意改变它，如设置为空。

There is no good reason to declare a field "public" and "static" without also declaring it "final". Most of the time this is a kludge to share a state among several objects. But with this approach, any object can do whatever it wants with the shared state, such as setting it to null.

Noncompliant Code Example

```
public class Greeter {
  public static Foo foo = new Foo();
  ...
}
```

Compliant Solution

```
public class Greeter {
  public static final Foo FOO = new Foo();
```

...

}

"ResultSet.isLast()" should not be used

有几个原因，可以说明为什么不要使用ResultSet.islast()。第一，支持这种方法只有前馈型结果集。其次，它可能是昂贵的（驱动程序可能需要取下一行来回答这个问题）。最后，尚不清楚当ResultSet是空的会返回什么，所以有些驱动程序可能会返回与预期相反的结论。

There are several reasons to avoid ResultSet.isLast(). First, support for this method is optional for TYPE_FORWARD_ONLY result sets. Second, it can be expensive (the driver may need to fetch the next row to answer the question). Finally, the specification is not clear on what should be returned when the ResultSet is empty, so some drivers may return the opposite of what is expected.

Noncompliant Code Example

```
stmt.executeQuery("SELECT name, address FROM PERSON");
ResultSet rs = stmt.getResultSet();
while (! rs.isLast()) { // Noncompliant
// process row
}
```

Compliant Solution

```
ResultSet rs = stmt.executeQuery("SELECT name, address FROM PERSON");
while (! rs.next()) {
// process row
}
```

"static final" arrays should be "private"

公共数组，即使是声明静态的、最终的，仍然可以被恶意程序改写其内容。final关键字在一个数组的声明意味着，数组对象本身可能只被分配一次，但其内容仍然是可变的。因此定义一个公共数组是一种安全风险。

相反，数组应该是私有的，通过方法访问。

Public arrays, even ones declared static final can have their contents edited by malicious programs. The final keyword on an array declaration means that the array object itself may only be assigned once, but its contents are still mutable. Therefore making arrays public is a security risk.

Instead, arrays should be private and accessed through methods.

"Threads" should not be used where "Runnables" are expected

虽然在一个需要使用runnable的地方使用thread并没有什么错，但是，两者的语义并不同，混合使用是一种不好的做法，可能会导致未来很头痛。

所以在实现多线程的过程中，public Thread(Runnable targer)这里不要用thread来代替runnable.

While it is technically correct to use a Thread where a Runnable is called for, the semantics of the two objects are different, and mixing

them is a bad practice that will likely lead to headaches in the future.

The crux of the issue is that Thread is a larger concept than Runnable. A Runnable is an object whose running should be managed. A Thread expects to manage the running of itself or other Runnables.

Noncompliant Code Example

```
public static void main(String[] args) {
    Thread r =new Thread() {
        int p;
        @Override
        public void run() {
            while(true)
                System.out.println("a");
        }
    };
    new Thread(r).start();  // Noncompliant
```

Compliant Solution

```
public static void main(String[] args) {
    Runnable r =new Runnable() {
        int p;
        @Override
        public void run() {
            while(true)
                System.out.println("a");
        }
    };
    new Thread(r).start();
```

Classes should not be compared by name

并没有要求说类名称必须是唯一的，只有它们在同一个包内不重名。因此，试图通过类名来确定一个对象的类型是一个充满危险的举动。其中一个危险是，恶意用户会发送与受信任的类同名的对象，从而得到信任的访问。

相反，instanceof运算符应用于检查对象的基本类型。

There is no requirement that class names be unique, only that they be unique within a package. Therefore trying to determine an object's type based on its class name is an exercise fraught with danger. One of those dangers is that a malicious user will send objects of the same name as the trusted class and thereby gain trusted access.

Instead, the instanceof operator should be used to check the object's underlying type.

Noncompliant Code Example

```
package computer;
class Pear extends Laptop { ... }

package food;
class Pear extends Fruit { ... }

class Store {

  public boolean hasSellByDate(Object item) {
    if ("Pear".equals(item.getClass().getSimpleName())) {  // Noncompliant
      return true;  // Results in throwing away week-old computers
    }
  }
}
```

Compliant Solution

```
class Store {

  public boolean hasSellByDate(Object item) {
    if (item instanceof food.Pear) {
      return true;
    }
  }
}
```

Cookies should be "secure"

“安全”属性用于防止cookie被HTTP明文连接发送的，那样很容易被窃听。相反，添加了安全属性的cookie只能通过加密的HTTPS连接发送。

The "secure" attribute prevents cookies from being sent over plaintext connections such as HTTP, where they would be easily eavesdropped upon. Instead, cookies with the secure attribute are only sent over encrypted HTTPS connections.

Noncompliant Code Example

```
Cookie c = new Cookie(SECRET, secret);  // Noncompliant; cookie is not secure
```

response.addCookie(c);

Compliant Solution

Cookie c = new Cookie(SECRET, secret);

c.setSecure(true);

response.addCookie(c);

Dissimilar primitive wrappers should not be used with the ternary operator without explicit casting

不同的原始包装不能用没有显式转换的三元运算符

如果包装的原始值（例如整数和浮点数）是用在三元运算符中（例如，a?b:c），都将取消装箱和强制转到一个共同的类型，可能会导致意想不到的结果。为了避免这种情况，需要添加一个显式转换为兼容类型。

If wrapped primitive values (e.g. Integers and Floats) are used in a ternary operator (e.g. a?b:c), both values will be unboxed and coerced to a common type, potentially leading to unexpected results. To avoid this, add an explicit cast to a compatible type.

Noncompliant Code Example

Integer i = 123456789;

Float f = 1.0f;

Number n = condition ? i : f; // Noncompliant; i is coerced to float. n = 1.23456792E8

Compliant Solution

Integer i = 123456789;

Float f = 1.0f;

Number n = condition ? (Number) i : f; // n = 123456789

Exception handlers should preserve the original exception

异常处理程序应保留原始异常

当处理捕获异常时，应记录或抛出原始异常的消息和堆栈信息。

When handling a caught exception, the original exception's message and stack trace should be logged or passed forward.

Noncompliant Code Example

// Noncompliant - exception is lost

```
try { /* ... */ } catch (Exception e) { LOGGER.info("context"); }

// Noncompliant - exception is lost (only message is preserved)
try { /* ... */ } catch (Exception e) { LOGGER.info(e.getMessage()); }

// Noncompliant - exception is lost
try { /* ... */ } catch (Exception e) { throw new RuntimeException("context"); }
```

Compliant Solution

```
try { /* ... */ } catch (Exception e) { LOGGER.info(e); }

try { /* ... */ } catch (Exception e) { throw new RuntimeException(e); }

try {
  /* ... */
} catch (RuntimeException e) {
  doSomething();
  throw e;
} catch (Exception e) {
  // Conversion into unchecked exception is also allowed
  throw new RuntimeException(e);
}
```

Execution of the Garbage Collector should be triggered only by the JVM

执行垃圾收集器（GC）应仅由JVM触发

主动call System.gc() or
Runtime.getRuntime().gc()是一个坏主意，原因很简单：没有办法知道JVM究竟会做什么，因为不同的供应商，版本或选项的JVM做做出不同的响应：

整个应用程序在调用期间会被冻结吗？

JVM会简单地忽略这个call吗？

...

一个应用程序依赖于那些不可预测的方法也会是不可预测的，因此不建议使用。

垃圾收集器运行的任务应该完全由JVM决定。

Calling System.gc() or Runtime.getRuntime().gc() is a bad idea for a simple reason: there is no way to know exactly what will be done under the hood by the JVM because the behavior will depend on its vendor, version and options:

Will the whole application be frozen during the call?

Is the -XX:DisableExplicitGC option activated?

Will the JVM simply ignore the call?

...

An application relying on those unpredictable methods is also unpredictable and therefore broken.

The task of running the garbage collector should be left exclusively to the JVM.

Ints and longs should not be shifted by more than their number of bits-1

int型和long型不应该位移超过它们的位数-1

因为int型是32位的，所以左右移32位和左右移0位效果是一样的。所以左右移超过位数次是没有意义的。通常都是BUG。。。

Since an int is a 32-bit variable, shifting by more than (-)31 is confusing at best and an error at worst. Shifting an int by 32 is the same as shifting it by 0, and shifting it by 33 is the same as shifting it by 1.

Similarly, shifting a long by (-)64 is the same as shifting it by 0, and shifting it by 65 is the same as shifting it by 1.

Noncompliant Code Example

```
public int shift(int a) {
  return a << 48;
}
```

Compliant Solution

```
public int shift(int a) {
  return a << 16;
}
```

Throwable.printStackTrace(...) should not be called

Throwable.printStackTrace（ ）不应该被调用

Throwable.printStackTrace会打印一个Throwable及其堆栈信息到输出流里。

但我们更推荐去logger去打印throwables，因为它们有许多优点：

1.用户能够轻松地检索日志。

2.日志消息的格式是统一的，允许用户轻松浏览日志。

Throwable.printStackTrace(...) prints a throwable and its stack trace to some stream.

Loggers should be used instead to print throwables, as they have many advantages:

Users are able to easily retrieve the logs.

The format of log messages is uniform and allow users to browse the logs easily.

The following code:

```
try {
  /* ... */
} catch(Exception e) {
  e.printStackTrace();        // Noncompliant
}
```

should be refactored into:

```
try {
  /* ... */
} catch(Exception e) {
  LOGGER.log("context", e);   // Compliant
}
```

Throwable and Error should not be caught

throwable 和 error不应该被catch

throwable是JAVA中所有错误和异常的超类，error是所有错误的超类，这些error不应该被应用catch住。如果catch throwable或是error，会将OutOfMemoryError and InternalError也捕捉，这些错误不应该由应用来捕获。catch了也处理不了，这一类问题一但发生就必须尽快抛出，中止程序的运行，catch住会导致问题越积越严重。

Throwable is the superclass of all errors and exceptions in Java.

Error is the superclass of all errors, which are not meant to be caught by applications.

Catching either Throwable or Error will also catch OutOfMemoryError and InternalError, from which an application should not attempt to recover.

Noncompliant Code Example

```
try { /* ... */ } catch (Throwable t) { /* ... */ }
try { /* ... */ } catch (Error e) { /* ... */ }
```

Compliant Solution

```
try { /* ... */ } catch (RuntimeException e) { /* ... */ }
try { /* ... */ } catch (MyException e) { /* ... */ }
```

Thread.run() and Runnable.run() should not be called directly

thread.run runnable.run不应该被立即调用

thread.run的目的是为了在一个新启的单独的线程中执行代码，在当前函数中立即调用这些函数没有意义，因为他们的代码会在当前的线程中被执行。

为了达到预期的效果，应该改用thread.start方法。

The purpose of the Thread.run() and Runnable.run() methods is to execute code in a separate, dedicated thread. Calling those methods directly doesn't make sense because it causes their code to be executed in the current thread.

To get the expected behavior, call the Thread.start() method instead.

Noncompliant Code Example

```
Thread myThread = new Thread(runnable);
myThread.run(); // Noncompliant
```

Compliant Solution

```
Thread myThread = new Thread(runnable);
myThread.start(); // Compliant
```

The Object.finalize() method should not be called

object.finalize()方法不应该被调用

根据JAVA文档，object.finalize函数会被垃圾收集器（GC）调用，用于当一个对象没有任何依赖的情况下回收资源。显式的调用这个方法会破坏这个规则，而且没有必要。

According to the official javadoc documentation, this Object.finalize() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. Calling this method explicitly breaks this contract and so is misleading.

Noncompliant Code Example

```
public void dispose() throws Throwable {
```

```
  this.finalize();                    // Noncompliant
}
```

The Object.finalize() method should not be overriden

object.finalize()方法不应该被重写

object.finalize方法会被垃圾收集器（GC）调用，用于回收那些没有任何其它对象引用的对象。但是，这个回收的调用并不是最后一个引用删除后立即执行的，通常会有几微秒到几分钟的延迟。所以，当你需要回归某个对象的系统资源时，最好是不依赖于这个异步机制来处理它们。

This Object.finalize() method is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. But there is absolutely no warranty that this method will be called AS SOON AS the last references to the object are removed. It can be few microseconds to few minutes later. So when some system resources need to be disposed by an object, it's better to not rely on this asynchronous mechanism to dispose them.

Noncompliant Code Example

```
public class MyClass {
  ...
  protected void finalize() {
    releaseSomeResources();    // Noncompliant
  }
  ...
}
```

The Array.equals(Object obj) method should not be used

array.equals()方法不应该被调用

由于数组没有重写object.equals()，调用这个方法比较两个数组仅仅是比较两个数组的地址。这意味着array1.equals(array2) 相当于array1==array2.

然而，还是有一些开发者以为Array.equals()可以做更多，比方说比较一下数组的大小和内容是否相等。为了避免这种误解，我们最好用'==' 操作符或是 Arrays.equals(array1, array2)来代替。

Since arrays do not override Object.equals(), calling equals on two arrays is the same as comparing their addresses. This means that array1.equals(array2) is equivalent to array1==array2.

However, some developers might expect Array.equals(Object obj) to do more than a simple memory address comparison, comparing for instance the size and content of the two arrays. To prevent such a misunderstanding, the '==' operator or Arrays.equals(array1, array2) must always be used in place of the Array.equals(Object obj) method.

Noncompliant Code Example

```
if(array1.equals(array2)){...}
```

Compliant Solution

```
if(Arrays.equals(array1, array2)){...}
```

or

```
if(array1 == array2){...}
```

Switch cases should end with an unconditional "break" statement

switch语句应该无条件的带一个“break”语句

当一个switch块没有显式地结束时，它继续执行下面的语句。虽然这有时是故意的，但它往往是一个错误，会导致意想不到的行为。

When the execution is not explicitly terminated at the end of a switch case, it continues to execute the statements of the following case. While this is sometimes intentional, it often is a mistake which leads to unexpected behavior.

Noncompliant Code Example

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:  // Both 'doSomething()' and 'doSomethingElse()' will be executed. Is it on purpose ?
    doSomething();
  default:
    doSomethingElse();
    break;
}
```

Compliant Solution

```
switch (myVariable) {
  case 1:
    foo();
    break;
  case 2:
    doSomething();
    break;
  default:
```

```
    doSomethingElse();
    break;
}
```

Exceptions

This rule is relaxed in the following cases:

```
switch (myVariable) {
  case 0:                       // Empty case used to specify the same behavior for a group of cases.
  case 1:
    doSomething();
    break;
  case 2:                       // Use of return statement
    return;
  case 3:                       // Use of throw statement
    throw new IllegalStateException();
  default:                      // For the last case, use of break statement is optional
    doSomethingElse();
}
```

"Serializable" inner classes of non-serializable classes should be "static"

非序列化类内部的序列化类应该是静态的。

序列化非静态内部类会导致在序列化类外部也在尝试。如果外部类不可序列化，然后序列化将会失败，导致运行时错误。

使内部类静态（即“嵌套”）避免了这个问题，因此，内部类应该是静态的，如果可能的话。

Serializing a non-static inner class will result in an attempt at serializing the outer class as well. If the outer class is not serializable, then serialization will fail, resulting in a runtime error.

Making the inner class static (i.e. "nested") avoids this problem, therefore inner classes should be static if possible. However, you should be aware that there are semantic differences between an inner class and a nested one:

an inner class can only be instantiated within the context of an instance of the outer class.

a nested (static) class can be instantiated independently of the outer class.

Noncompliant Code Example

```
public class Pomegranate {
  // ...

  public class Seed implements Serializable {  // Noncompliant; serialization will fail
```

```
    // ...
  }
}
```

Compliant Solution

```
public class Pomegranate {
  // ...

  public static class Seed implements Serializable {
    // ...
  }
}
```

Credentials should not be hard-coded

凭据不应该硬编码

因为从编辑的应用中提取字符串是很容易的，所以所有的凭据（密码）都不应该硬编码（直接写在代码里）。

Because it is easy to extract strings from a compiled application, credentials should never be hard-coded. Do so, and they're almost guaranteed to end up in the hands of an attacker. This is particularly true for applications that are distributed.

Credentials should be stored outside of the code in a strongly-protected encrypted configuration file or database.

Noncompliant Code Example

```
Connection conn = null;
try {
  conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
      "user=steve&password=blue"); // Noncompliant
  String uname = "steve";
  String password = "blue";
  conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
      "user=" + uname + "&password=" + password); // Noncompliant
```

Compliant Solution

```
Connection conn = null;
try {
  String uname = getEncryptedUser();
  String password = getEncryptedPass();
  conn = DriverManager.getConnection("jdbc:mysql://localhost/test?" +
```

"user=" + uname + "&password=" + password);