



JAVA

STT	Tên yêu cầu
1	Không sử dụng "Double.longBitsToDouble" với tham số kiểu "int"
2	Không sử dụng "Lock" trong khối "synchronized"
3	Override hai phương thức "equals(Object obj)" và "hashCode()" cùng lúc
4	Kiểm tra tham số đầu vào đối với phương thức "equals(Object obj)"
5	Không để "return" trong các khối "finally"
6	Trong khối synchronized trên một đối tượng không thực hiện gán giá trị khác cho đối tượng đó.
7	Không gọi phương thức "wait(...)", "notify()", "notifyAll()" trong Thread
8	Kiểm tra giá trị Null
9	Đóng tài nguyên sau khi sử dụng
10	Điều kiện logic cần được đảm bảo để không truy cập đối tượng Null
11	Cài đặt interface "Cloneables" cần phải override phương thức "clone"
12	Nếu override phương thức "equals(Object obj)" hoặc "compareTo(T obj)" thì override cả hai phương thức.
13	Biến trong điều kiện dừng vòng for và biến thay đổi sau mỗi vòng lặp phải là cùng một biến
14	Biến trong điều kiện dừng vòng for không nên bị thay đổi bên trong nội dung vòng for hoặc phụ thuộc kết quả trả về từ đoạn code khác.
15	Khai báo final với các biến "public static"
16	Có từ khóa "case" trong mỗi khối của khai báo switch
17	Không truyền một đối tượng Collection vào method của chính đối tượng đó
18	Không throw exception trong Servlet
19	Việc dọn rác chỉ thực hiện từ JVM
20	Không so sánh bằng với dữ liệu kiểu Float
21	Không truy cập đến các thuộc tính static từ phương thức của Instance
22	Khởi tạo trường static đặt trong "synchronized"

23	Giải phóng Lock
24	Không khai báo "public static" với các trường có thể thay đổi
25	Không sử dụng "Math.abs" đối với giá trị có thể là "MIN_VALUE"
26	Không write các lớp non-serializable
27	Loại bỏ các điều kiện kiểm tra không bao giờ xảy ra trong if/else.
28	Không nên bỏ qua giá trị trả lại của hàm.
29	Không dùng các toán tử không phải short-circuit trong các biểu thức điều kiện (condition)
30	Kết thúc Switch cases bằng lệnh "break"
31	Không gọi trực tiếp Thread.run() và Runnable.run()
32	Không gọi Throwable.printStackTrace(...)
33	Khi xử lý exception cần bảo lưu exception ban đầu
34	Các biểu thức điều kiện không nên luôn luôn là "TRUE" hoặc "FALSE"
35	Không khai báo biến static với đối tượng kiểu "Calendars" và "DateFormats"
36	Không so sánh kiểu Class bằng Class Name
37	Không sử dụng cùng một toán tử trên 2 vế của một biểu thức nhị phân
38	Loại bỏ các "dead store"
39	Khi kiểm tra điều kiện bằng với một biến String, chuỗi Strings nên được đặt ở bên trái của biểu thức so sánh equal
40	Sử dụng ConcurrentHashMap thay cho HashMap
41	Sử dụng pool khi gọi webservice
42	Đảm bảo hiệu năng thao tác cập nhật dữ liệu lên redis sử dụng thư viện jedis
43	Lưu ý khi sử dụng synchronized

### 1. Không sử dụng "Double.longBitsToDouble" với tham số kiểu "int"

Double.longBitsToDouble yêu cầu đối số là kiểu long 64bit, vì vậy khi chuyển số nhỏ như int sang dạng double sẽ có thể gây ra lỗi do việc bố trí các bit không đúng.

Ví dụ code không tuân thủ

```
int i = 42;
double d = Double.longBitsToDouble(i); // Không tuân thủ do i là kiểu int
```

Cách viết đúng:

```
long i = 42;
double d = Double.longBitsToDouble(i);
```

### 2. Không sử dụng "Lock" trong khối "synchronized"

Java.util.concurrent.locks cung cấp các phương thức khóa mềm dẻo và mạnh hơn so với khối synchronized, sử dụng synchronize với đối tượng Lock sẽ làm mất ưu điểm này

Ví dụ code không tuân thủ

```
Lock lock = new MyLockImpl();
synchronized(lock) { // không synchronized với đối tượng kiểu Lock
    //...
}
```

Cách viết đúng:

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

### 3. Override hai phương thức "equals(Object obj)" và "hashCode()" cùng lúc

Theo đặc tả ngôn ngữ Java, có ràng buộc giữa 2 phương thức equals(Object) và hashCode():

- Nếu 2 đối tượng bằng nhau theo phương thức equals(Object) thì khi gọi phương thức hashCode với mỗi đối tượng phải trả về cùng kết quả là số nguyên
- Nếu 2 đối tượng không bằng nhau theo phương thức equals(Object) thì khi hashCode sẽ trả về các kết quả là các số nguyên riêng biệt

Theo như ràng buộc này, thì 2 phương thức nên cùng được override khi sử dụng

Ví dụ code không tuân thủ



```
class MyClass {    //chua override "hashCode()"

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

}
```

Cách viết đúng:

```
class MyClass {

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

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

}
```

#### 4. Kiểm tra tham số đầu vào đối với phương thức "equals(Object obj)"

Phương thức “equals” sử dụng Object làm tham số đầu vào do vậy bất kỳ đối tượng nào đều có thể được truyền vào để so sánh, không nên mặc định rằng chỉ đối tượng cùng kiểu được truyền vào mà cần kiểm tra để đảm bảo không có lỗi xảy ra.

Ví dụ code không tuân thủ

```
public boolean equals(Object obj) {
    MyClass mc = (MyClass)obj;    // đối tượng obj truyền vào có thể null hoặc
    thuộc kiểu đối tượng khác MyClass
    // ...
}
```

Cách viết đúng:

```
public boolean equals(Object obj) {
    if (obj == null)
        return false;

    if (this.getClass() != obj.getClass())
        return false;

    MyClass mc = (MyClass)obj;
    // ...
}
```

## 5. Không để "return" trong các khối "finally"

Không được gọi return trong khối finally khi xử lý ngoại lệ, nếu không các ngoại lệ sẽ không được throw trong các khối khối try hoặc catch() nếu có

Ví dụ code không tuân thủ

```
public static void main(String[] args) {
    try {
        doSomethingWhichThrowsException();
        System.out.println("OK"); // message này vẫn được hiển thị mặc dù theo
        // logic câu lệnh print không được thực hiện do phương thức trên throw exception
        // và chuyển tiếp vào trong khối catch
    } catch (RuntimeException e) {
        System.out.println("ERROR"); // message này sẽ không được hiển thị
    }
}

public static void doSomethingWhichThrowsException() {
    try {
        throw new RuntimeException();
    } finally {
        /* ... */
        return; // khai báo return ở đây sẽ làm cho throw trong khối try bên
        // trên sẽ không được thực hiện
    }
}
```

Cách viết đúng:

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

## 6. Trong khối synchronized trên một đối tượng không thực hiện gán giá trị khác cho đối tượng đó.

Việc synchronized trên một đối tượng thực chất là synchronized trên một thể hiện (object instance) được gán cho đối tượng. Gán giá trị khác cho đối tượng đó trong khối synchronized sẽ làm cho khối này có thể được chạy bởi các thread khác.

Ví dụ code không tuân thủ

```
private String color = "red";

private void doSomething() {
    synchronized(color) { // lock thực chất được thực hiện trên "red" được
        //...
        color = "green"; //lỗi, sau khi thực hiện phép gán các thread khác sẽ
        // được phép chạy khối synchronized này
        // ...
    }
}
```

Cách viết đúng:

```
private String color = "red";
private Object lockObj = new Object();

private void doSomething() {
    synchronized(lockObj) {
        //...
        color = "green";
        // ...
    }
}
```

## 7. Không gọi phương thức "wait(...)", "notify()", "notifyAll()" trong Thread

Không gọi các phương thức này khi sử dụng Thread do JVM dựa vào các phương thức này để thay đổi trạng thái của Thread (BLOCKED, WAITING,...), việc gọi chúng sẽ làm sai hành vi của JVM.

Ví dụ code không tuân thủ

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

## 8. Kiểm tra giá trị Null

Khi truy cập nội dung trong đối tượng NULL thì chương trình sẽ xuất hiện lỗi NullPointerException, chương trình có thể gặp lỗi nghiệp vụ hoặc bị dừng giữa chừng,

nặng hơn, hacker có thể lợi dụng để tấn công hệ thống. Cần thực hiện kiểm tra đối tượng khác Null trước khi truy cập.

Chú ý chúng ta có thể sử dụng annotations: `@CheckForNull` và `@Nonnull` để chỉ ra những giá trị Null hoặc không Null.

`@Nullable` chỉ ra rằng trong một vài trường hợp có thể nhận giá trị Null.

Ví dụ code không tuân thủ

```
@CheckForNull
String getName() {...}

public boolean isEmpty() {
    return getName().length() == 0; // Giá trị hàm getName() chưa được kiểm
    tra khác Null
}

Connection conn = null;
Statement stmt = null;
try {
    conn = DriverManager.getConnection(DB_URL,USER,PASS);
    stmt = conn.createStatement();
    // ...
} catch(Exception e) {
    e.printStackTrace();
} finally {
    stmt.close(); // stmt chưa được kiểm tra khác Null
    conn.close(); // conn chưa được kiểm tra khác Null
}

private void merge(@Nonnull Color firstColor, @Nonnull Color
secondColor){...}

public void append(@CheckForNull Color color) {
    merge(currentColor, color); // color nên được kiểm tra khác Null vì hàm
    merge không chấp nhận giá trị Null
}

void paint(Color color) {
    if(color == null) {
        System.out.println("Unable to apply color " + color.toString()); //
        color chưa được kiểm tra khác Null
        return;
    }
    ...
}
```



## 9. Đóng tài nguyên sau khi sử dụng

Sau khi sử dụng cần thực hiện đóng để giải phóng tài nguyên cho các tiến trình khác sử dụng và giải phóng bộ nhớ.

Ví dụ code không tuân thủ

```
OutputStream stream = null;
try{
    for (String property : propertyList) {
        stream = new FileOutputStream("myfile.txt");//Mở nhiều stream
        // ...
    }
}catch(Exception e){
    // ...
}finally{
    stream.close(); //Nhiều stream được mở, nhưng chỉ cái cuối cùng được đóng
}
```

Cách viết đúng:

```
OutputStream stream = null;
try{
    stream = new FileOutputStream("myfile.txt");
    for (String property : propertyList) {
        // ...
    }
}catch(Exception e){
    // ...
}finally{
    stream.close();
}

//Java 7 giới thiệu một cách viết code giúp đóng tất cả các đối tượng trong
câu lệnh try như sau:

try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
    //...
}
catch ( ... ) {
    //...
}
```

## 10. Điều kiện logic cần được đảm bảo để không truy cập đối tượng Null

Khi viết điều kiện logic ta cần chú ý điều kiện nào được thực hiện trước, điều kiện nào được thực hiện sau để đảm bảo không xảy ra trường hợp truy cập đối tượng Null

Ví dụ code không tuân thủ

```
if (str == null && str.length() == 0) {
    System.out.println("String is empty"); //cả 2 điều kiện str == null và
```

```
str.length() == 0 đều được thực hiện, khi đó nếu str null thì str.length()
sẽ gây ra lỗi
}

if (str != null || str.length() > 0) {
    System.out.println("String is not empty");
}
```

Cách viết đúng:

```
if (str == null || str.length() == 0) {
    System.out.println("String is empty"); //nếu str là null thì điều kiện
    str.length() sẽ không được thực hiện và không gây ra lỗi
}

if (str != null && str.length() > 0) {
    System.out.println("String is not empty");
}
```

## 11. Cài đặt interface "Cloneables" cần phải override phương thức "clone"

Cài đặt interface Cloneables thì cần phải override phương thức clone, nếu không phương thức mặc định JVM clone sẽ được dùng khi đó chỉ các thuộc tính nguyên thủy (primitive) được sao chép, đối với các thuộc tính khác chỉ copy tham chiếu (reference) sang đối tượng clone, đối tượng được clone có thể sẽ dùng chung thuộc tính với đối tượng nguồn.

Ví dụ code không tuân thủ

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

Cách viết đúng:

```
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();
        //...
    }
}
```

## 12. Nếu override phương thức "equals(Object obj)" hoặc "compareTo(T obj)" thì override cả hai phương thức.

Theo tài liệu Java về phương thức Comparable.compareTo(T o): khuyến khích nhưng không bắt buộc việc đảm bảo (x.compareTo(y)==0) tương đương với (x.equals(y)).

Do đó để tránh nhầm lẫn về sau, nếu override phương thức equals() hoặc compareTo() thì nên override cả hai phương thức cùng nhau.

Ví dụ code không tuân thủ

```
public class Foo implements Comparable<Foo> {
    @Override
    public int compareTo(Foo foo) { /* ... */ } // phương thức equals(Object
obj) không được override
}
```

Cách viết đúng:

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

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

## 13. Biến trong điều kiện dừng vòng for và biến thay đổi sau mỗi vòng lặp phải là cùng một biến

Biến trong điều kiện dừng vòng for và biến thay đổi sau mỗi vòng lặp phải là cùng một biến, nếu không có thể dẫn đến vòng lặp không bao giờ kết thúc.

Ví dụ code không tuân thủ

```
for (i = 0; i < 10; j++) { //điều kiện dừng vòng lặp dựa trên biến i, tuy
nhiên i không được tăng trong về cuối của for
// ...
}
```

Cách viết đúng:

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

#### 14. Không thay đổi biến trong điều kiện dừng vòng for bên trong nội dung vòng lặp hoặc phụ thuộc kết quả trả về từ đoạn code khác.

Biến trong điều kiện dừng vòng for không nên bị thay đổi bên trong vòng lặp hoặc phụ thuộc kết quả trả về từ đoạn code khác. Việc này có thể dẫn đến nguy cơ vòng lặp không bao giờ kết thúc.

Ví dụ code không tuân thủ

```
for (int i = 0; i < 10; i++) {  
    ...  
    i = i - 1; // Biến đếm bị thay đổi giá trị bên trong vòng lặp  
    ...  
}  
  
for (int i = 0; i < getMaximumNumber(); i++) { ... } // Điều kiện dừng vòng  
lặp phụ thuộc vào kết quả trả về từ hàm getMaximumNumber()
```

Cách viết đúng:

```
int stopCondition = getMaximumNumber();  
for (int i = 0; i < stopCondition; i++) { ... }
```

#### 15. Khai báo final với biến "public static"

Không có lý do gì để khai báo một biến là "public" và "static" mà không "final". Như thế bất kỳ ở đâu cũng có thể thay đổi giá trị này và có thể gây lỗi chương trình.

Ví dụ code không tuân thủ

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

Cách viết đúng:

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

#### 16. Cờ từ khóa "case" trong mỗi khối của khai báo switch

Đôi khi việc khai báo một khối lệnh trong switch mà không bắt đầu bằng từ khóa case vẫn hợp lệ, tuy nhiên việc này làm cho chương trình khó hiểu và thường là do lỗi typing.

Ví dụ code không tuân thủ

```
switch (day) {  
    case MONDAY:
```



```

case TUESDAY:
WEDNESDAY: //đúng về cú pháp nhưng có thể dẫn đến hành vi bất thường
             không kiểm soát được
    doSomething();
    break;
    ...
}

```

Cách viết đúng:

```

switch (day) {
case MONDAY:
    break;
case TUESDAY:
    foo();
    break;
}

```

## 17. Không truyền một đối tượng Collection vào method của chính đối tượng đó

Việc này có thể gây lỗi chương trình.

Ví dụ code không tuân thủ

```

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

objs.add(objs); // không tuân thủ; gây ra exception StackOverflowException
nếu objs.hashCode() được gọi
objs.containsAll(objs); // không tuân thủ; luôn luôn trả về true
objs.removeAll(objs); // không tuân thủ; dễ gây nhầm lẫn. thay vào đó nên
sử dụng clear()
.....

```

## 18. Không throw exception trong Servlet

Việc throw các exception trong servlet có thể đưa web server vào các trạng thái không mong muốn, có khả năng bị tấn công bởi hình thức từ chối dịch vụ.

Ví dụ code không tuân thủ

```

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    String ip = request.getRemoteAddr();
    InetAddress addr = InetAddress.getByName(ip); // Noncompliant;
    getByName(String) throws UnknownHostException
    //....
}

```

Cách viết đúng:

```

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {
    try {

```

```
String ip = request.getRemoteAddr();
InetAddress addr = InetAddress.getByName(ip);
//...
}
catch (UnknownHostException uhex) {
//...
}
```

## 19. Việc dọn rác chỉ thực hiện từ JVM

Không nên gọi `System.gc()` hoặc `Runtime.getRuntime().gc()` bởi vì không thể biết chính xác JVM sẽ thực hiện ngầm những gì, điều này phụ thuộc vào nhà cung cấp, phiên bản và các tùy chọn:

- Toàn bộ ứng dụng có bị đóng băng khi đang gọi các phương thức này không?
- Tùy chọn `-XX:DisableExplicitGC` có được kích hoạt không?
- JVM sẽ bỏ qua việc gọi các hàm này một cách tường minh?
- ...

Nhiệm vụ thu gom rác nên để dành riêng cho JVM

## 20. Không so sánh bằng với dữ liệu kiểu Float

Các phép toán trên số Float là không chính xác. Thậm chí khi thực hiện một dãy các phép toán trên số Float, mỗi lần chạy cho một kết quả khác nhau, nó phụ thuộc vào Compiler và setting của Compiler.

Ví dụ code không tuân thủ

```
float zeroFloat = 0.0f;
if (zeroFloat == 0) { // Không đúng qui luật, trả về false
}
```

Cách viết đúng:

```
float zeroFloat = 0.0f;
if (Float.floatToRawIntBits(zeroFloat) == 0) { //Đúng qui luật. Sử dụng so
sánh bit đảm bảo chúng ta so sánh được với giá trị 0
}
```

## 21. Không truy cập đến các thuộc tính static từ phương thức của Instance

Cập nhật một trường static từ một phương thức không static có thể dễ dẫn đến lỗi nếu có nhiều Instance của Lớp hoặc nhiều Thread đang chạy. Lý tưởng nhất là các trường static chỉ được cập nhật từ các phương thức static và synchronize.

Ví dụ code không tuân thủ

```
public class MyClass {
    private static int count = 0;

    public void doSomething() {
        //...
        count++; // không tuân thủ, thay đổi thuộc tính static
    }
}
```

## 22. Khởi tạo trường static đặt trong "synchronized"

Trong tình huống xử lý đa luồng có thể xảy ra trường hợp tiến trình thứ hai truy cập một đối tượng đang được khởi tạo dở bởi tiến trình đầu tiên. Cho phép truy cập như vậy có thể gây ra lỗi nghiêm trọng. Giải pháp là khởi tạo cho trường static nên được synchronized hoặc khai báo biến là volatile.

Ví dụ code không tuân thủ

```
protected static Object instance = null;

public static Object getInstance() {
    if (instance != null) {
        return instance;
    }

    instance = new Object(); //Không đúng quy luật
    return instance;
}
```

Cách viết đúng:

```
protected static volatile Object instance = null;

public static Object getInstance() {
    if (instance != null) {
        return instance;
    }

    instance = new Object();
    return instance;
}

//hoặc

protected static Object instance = null;

public static synchronized Object getInstance() {
    if (instance != null) {
        return instance;
    }
}
```

```
instance = new Object();  
return instance;  
}
```

### 23. Giải phóng Lock

Các logic trong một phương thức cần đảm bảo rằng Lock được giải phóng trong các phương thức gọi nó. Thất bại trong việc giải phóng Lock làm tăng nguy cơ DeadLock và có thể gây ra lỗi runtime.

Ví dụ code không tuân thủ

```
public class MyClass {  
    Lock lock = new Lock();  
  
    public void acquireLock() {  
        lock.lock(); // không tuân thủ, cần release ngay trong phương thức này.  
    }  
  
    public void releaseLock() {  
        lock.unlock();  
    }  
  
    public void doTheThing() {  
        acquireLock();  
        // do work...  
        releaseLock();  
    }  
}
```

Cách viết đúng:

```
public class MyClass {  
    Lock lock = new Lock();  
  
    public void doTheThing() {  
        lock.lock();  
        // do work...  
        lock.unlock();  
    }  
}
```

### 24. Không khai báo "public static" các trường có thể thay đổi

Đối với các trường có thể thay đổi không nên khai báo public static. Trường đó phải được chuyển vào class để giảm sự truy cập trực tiếp. Tránh nguy cơ lỗi khi nhiều tiến trình cùng thay đổi giá trị.

Ví dụ code không tuân thủ

```
public interface MyInterface {
```



```

public static String [] strings; // không tuân thủ
}

public class A {
    public static String [] strings1 = {"first","second"}; // không tuân thủ
    public static String [] strings2 = {"first","second"}; // không tuân thủ
    public static List<String> strings3 = new ArrayList<>(); // không tuân
    thủ
    // ...
}

```

## 25. Không sử dụng "Math.abs" đối với giá trị có thể là "MIN\_VALUE"

Có khả năng hashCode trả về giá trị Integer.MIN\_VALUE, trị tuyệt đối của giá trị này vẫn có thể là giá trị âm trong khi giá trị mong đợi là một số dương. Điều này dẫn đến kết quả bất thường, không tin cậy được.

Ví dụ code không tuân thủ

```

public void doSomething(String str) {
    if (Math.abs(str.hashCode()) > 0) { // không tuân thủ do str.hashCode() có
    thể là MIN_VALUE
        // ...
    }
}

```

Cách viết đúng:

```

public void doSomething(String str) {
    if (str.hashCode() != 0) {
        // ...
    }
}

```

## 26. Không thực hiện write các lớp non-serializable

Thực hiện thao tác này có thể dẫn đến exception.

Ví dụ code không tuân thủ

```

public class Vegetable { // không implement Serializable hoặc extends từ
class cha có implement Serializable
    //...
}

public class Menu {
    public void meal() throws IOException {
        Vegetable veg;
        //...
        FileOutputStream fout = new FileOutputStream(veg.getName());
        ObjectOutputStream oos = new ObjectOutputStream(fout);
        oos.writeObject(veg); // Không đúng, không có gì được ghi ra file
    }
}

```

Cách viết đúng:

```
public class Vegetable implements Serializable { // đối tượng này đã có thể
    được serialize
    //...
}

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

## 27. Loại bỏ các điều kiện kiểm tra không bao giờ xảy ra trong if/else.

Câu lệnh if/else if/else được kiểm tra từ trên xuống dưới, vì vậy tối đa chỉ có 1 nhánh được thực hiện (khi điều kiện kiểm tra là true). Vì vậy, nếu điều kiện kiểm tra trùng nhau hoặc không bao giờ xảy ra sẽ làm cho code khó hiểu, mất thời gian kiểm tra khi chạy chương trình.

Ví dụ code không tuân thủ

```
if (param == 1) // Kiểm tra lần 1
    openWindow();
else if (param == 2)
    closeWindow();
else if (param == 1) // Không bao giờ chạy đến đây hoặc luôn luôn có giá
    trị là false
    moveWindowToTheBackground();
else if (param == 1 && param == 2) // Không bao giờ chạy đến đây hoặc luôn
    luôn có giá trị là false
    doSomethings();
}
```

Cách viết đúng:

```
if (param == 1)
    openWindow();
else if (param == 2)
    closeWindow();
else if (param == 3)
    moveWindowToTheBackground();
}
```

## 28. Không nên bỏ qua giá trị trả lại của hàm.

Kiểm tra các đoạn mã không ảnh hưởng đến toàn bộ chương trình. Nếu thừa thì tiến hành loại bỏ.

Ví dụ code không tuân thủ

```
public void handle(String command){
    command.toLowerCase(); // Noncompliant; result of method thrown away
    ...
}
```

Cách viết đúng:

```
public void handle(String command){
    String formattedCommand = command.toLowerCase();
    ...
}
```

## 29. Không dùng các toán tử không phải dạng short-circuit trong các biểu thức điều kiện (condition)

Việc sử dụng phép toán logic không phải dạng short-circuit (||, &&) trong biểu thức điều kiện có thể gây ra lỗi nghiêm trọng trong chương trình.

Ví dụ code không tuân thủ

```
if(getTrue() | getFalse()) { ... } // cả 2 điều kiện đều được đánh giá
```

Cách viết đúng:

```
if(getTrue() || getFalse()) { ... }
```

## 30. Kết thúc Switch cases bằng lệnh "break"

Kết thúc một case nếu không break thì case tiếp theo sẽ được tiếp tục thực hiện, vì vậy có thể gây ra lỗi không kiểm soát được

Ví dụ code không tuân thủ

```
switch (myVariable) {
    case 1:
        foo();
        break;
    case 2: // 'doSomething()' và 'doSomethingElse()' đều sẽ được thực hiện,
           đây là chủ ý hay do lỗi typing?
        doSomething();
    default:
        doSomethingElse();
        break;
}
```

Cách viết đúng:

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

//Luật này có thể không áp dụng trong trường hợp sau:

switch (myVariable) {
    case 0: // case rỗng, được sử dụng để đại
    diện cho một nhóm các case có chung hành vi
    case 1:
        doSomething();
        break;
    case 2: // sử dụng return
        return;
    case 3: // sử dụng return
        throw new IllegalStateException();
    default: // đối với case cuối cùng, không
    bắt buộc phải có break
        doSomethingElse();
}
```

### 31. Không gọi trực tiếp Thread.run() và Runnable.run()

Việc gọi các phương thức này một cách trực tiếp sẽ không có ý nghĩa vì nó sẽ được thực hiện ngay trên thread hiện tại. Sử dụng phương thức Thread.start() để thay thế

Ví dụ code không tuân thủ

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

Cách viết đúng:

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

### 32. Không gọi Throwable.printStackTrace(...).

Không nên sử dụng Throwable.printStackTrace(...) để in log ra màn hình (chỉ sử dụng trong trường hợp debug). Thay vào đó nên sử dụng các thư viện ghi log tiện lợi hơn để tận dụng các ưu điểm:



- Có thể dễ dàng lấy lại logs
- Định dạng các thông báo log thống nhất.

Ví dụ code không tuân thủ

```
try {
    /* ... */
} catch (Exception e) {
    e.printStackTrace(); //Không tuân thủ luật
}
```

Cách viết đúng:

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

### 33. Khi xử lý exception cần bảo lưu exception ban đầu

Khi xử lý một exception bắt được, message và stack trace của exception gốc cần được ghi log và forward.

Ví dụ code không tuân thủ

```
try { /* ... */ } catch (Exception e) { LOGGER.info("context"); }
//exception không được bảo lưu

try { /* ... */ } catch (Exception e) { LOGGER.info(e.getMessage()); }
//exception không được bảo lưu (only chỉ bảo lưu được message)

try { /* ... */ } catch (Exception e) { throw new
RuntimeException("context"); } //exception không được bảo lưu
```

Cách viết đúng:

```
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) {
    throw new RuntimeException(e); // được phép convert exception
}

//Ngoại lệ
```

```

InterruptedException, NumberFormatException, ParseException và
MalformedURLException

int myInteger;
try {
    myInteger = Integer.parseInt(myString);
} catch (NumberFormatException e) {
    // hoàn toàn có thể chấp nhận được chỉ k không xử lý "e" ở chỗ này
    myInteger = 0;
}

```

#### 34. Các biểu thức điều kiện không nên luôn luôn là "TRUE" hoặc "FALSE"

Biểu thức điều kiện luôn luôn FALSE làm cho khối lệnh tiếp theo không bao giờ được gọi, tương tự nếu luôn luôn TRUE nghĩa là mệnh đề điều kiện đang bị thừa và làm code trở nên khó đọc. Cần loại bỏ các biểu thức điều kiện này hoặc điều chỉnh lại để không xảy ra tình trạng luôn luôn TRUE hoặc FALSE

Ví dụ code không tuân thủ

```

//foo không thể đồng thời bằng và không bằng bar trong cùng một biểu thức
điều kiện
if(foo == bar && something && foo != bar) {...}
private void compute(int foo) {
    if (foo == 4) {
        doSomething();
        // foo bằng 4, do đó điều kiện này luôn luôn false
        if (foo > 4) {...}
    }
    ...
}
private void compute(boolean foo) {
    if (foo) {
        return;
    }
    doSomething();
    // ở chỗ này foo luôn luôn false
    if (foo){...}
    ...
}

```

#### 35. Không khai báo biến static với đối tượng kiểu "Calendars" và "DateFormats"

Calendar và DateFormat không đảm bảo thread-safe. Sử dụng các đối tượng này khi xử lý đa luồng dễ dẫn đến vấn đề về dữ liệu hoặc exceptions khi chạy runtime.

Ví dụ code không tuân thủ

```

public class MyClass {
    static private SimpleDateFormat format = new SimpleDateFormat("HH-mm-ss");
}

```

```
// Không tuân thủ
static private Calendar calendar = Calendar.getInstance(); // Không tuân
thủ
```

Cách viết đúng:

```
public class MyClass {
    private SimpleDateFormat format = new SimpleDateFormat("HH:mm:ss");
    private Calendar calendar = Calendar.getInstance();
}
```

### 36. Không so sánh kiểu Class bằng Class Name

Không có ràng buộc nào khiến cho class name là duy nhất (thực tế chỉ duy nhất trong 1 package), do đó xác định kiểu object dựa trên class name là một việc làm nguy hiểm và có thể dẫn đến lỗi chương trình. Một trong những nguy cơ là kẻ tấn công có thể gửi đến các đối tượng có cùng tên với các class tin cậy (trusted) và qua đó đạt được các truy cập dành riêng cho class tin cậy.

Thay vào đó sử dụng toán tử instanceof để kiểm tra kiểu object.

Ví dụ code không tuân thủ

```
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())) { // không tuân thủ
            return true;
        }
    }
}
```

Cách viết đúng:

```
class Store {

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

### 37. Không sử dụng cùng một toán tử trên 2 vế của một biểu thức nhị phân

Xảy ra trường hợp này thường là do nhầm lẫn (copy/paste) hoặc đơn giản là code thừa

Luận này không áp dụng với \*, +, và =

Ví dụ code không tuân thủ

```
if ( a == a ) { // luôn luôn true
doZ();
}
if ( a != a ) { // luôn luôn false
doY();
}
if ( a == b && a == b ) { // nếu về đầu tiên là true, thì về thứ 2 cũng vậy
doX();
}
if ( a == b || a == b ) { // nếu về đầu tiên là true, thì về thứ 2 cũng vậy
doW();
}

int j = 5 / 5; //luôn luôn là 1
int k = 5 - 5; //luôn luôn là 0
```

Cách viết đúng:

```
doZ();

if ( a == b ) {
doX();
}
if ( a == b ) {
doW();
}

int j = 1;
int k = 0;

//Ngoại lệ

//So sánh một số float với chính nó để kiểm tra giá trị NaN
//Tuong tự dịch 1 bit đối với 1 là cách thông dụng để tạo bit masks.

float f;
if(f != f) { //kiểm tra giá trị NaN
System.out.println("f is NaN");
}

int i = 1 << 1; // Hợp lệ
int j = a << a; // không hợp lệ
```

### 38. Loại bỏ các "dead store"

Dead store là một trong hai trường hợp sau:

- Khai báo một biến local và gán giá trị cho biến đó (kể cả giá trị null), tuy nhiên sau đó không được sử dụng ở bất kỳ chỗ



- Tính toán và lấy ra một giá trị nhưng sau đó không dùng.

Hai trường hợp này thường có thể dẫn đến lỗi nghiêm trọng, thậm chí không gây ra lỗi cũng làm tốn tài nguyên không cần thiết.

Ví dụ code không tuân thủ

```
public void pow(int a, int b) {
    if(b == 0) {
        return 0;
    }
    int x = a;
    for(int i= 1, i < b, i++) {
        x = x * a; //giá trị này sau đó không dùng ở đâu cả
    }
    return a;
}
```

Cách viết đúng:

```
public void pow(int a, int b) {
    if(b == 0) {
        return 0;
    }
    int x = a;
    for(int i= 1, i < b, i++) {
        x = x * a;
    }
    return x;
}
```

39. Khi kiểm tra điều kiện bằng với một biến String, chuỗi Strings nên được đặt ở bên trái của biểu thức so sánh equal

Nên đặt chuỗi String ở về bên trái của phương thức equals() hoặc equalsIgnoreCase(), như thế có thể ngăn các lỗi null pointer exceptions xảy ra do một chuỗi string sẽ luôn luôn là khác null.

Ví dụ code không tuân thủ

```
String myString = null;

System.out.println("Equal? " + myString.equals("foo"));
// Lỗi null pointer exception
System.out.println("Equal? " + (myString != null &&
myString.equals("foo"))); // cách viết rườm rà, phức tạp
```

Cách viết đúng:

```
System.out.println("Equal?" + "foo".equals(myString)); // ngắn gọn, giải
quyết được trường hợp null
```

#### 40. Sử dụng ConcurrentHashMap thay cho HashMap

Từ Java5, ConcurrentHashMap được cài đặt, thiết kế đặc biệt hướng tới các ứng dụng xử lý song song, đa luồng. Nên sử dụng thay thế cho HashMap hoặc các class implement interface Map.

#### 41. Sử dụng pool khi gọi webservice

Khi ứng dụng java gọi webservice của một hệ thống khác cần sử dụng pool

Ví dụ sử dụng PoolingHttpClientConnectionManager trong thư viện HttpClient

Tạo class HttpClientFactory để quản lý

```
import org.apache.http.client.config.RequestConfig;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.impl.conn.PoolingHttpClientConnectionManager;

import javax.net.ssl.SSLContext;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author
 * @created Feb 11, 2017
 */
public class HttpClientFactory {

    private static volatile PoolingHttpClientConnectionManager cm ;

    private static void initPool() throws Exception{
        if(cm==null){
            cm = new PoolingHttpClientConnectionManager();
            cm.setMaxTotal(300);
            cm.setDefaultMaxPerRoute(100);
        }
    }

    static{
        try {
            initPool();
        } catch (Exception ex) {
            Logger.getLogger(HttpClientFactory.class.getName()).log(Level.SEVERE,
            null, ex);
        }
    }

    public static CloseableHttpClient getHttpClientFromPool() throws Exception {
        RequestConfig requestConfig = RequestConfig.custom()
            .setConnectTimeout(30000)
    }
```

```

        .setSocketTimeout(30000).build();
        CloseableHttpClient client = HttpClients.custom()
            .setConnectionManager(cm)
            .setDefaultRequestConfig(requestConfig)
            .setConnectionManagerShared(true)
            .disableAutomaticRetries()
            .build();

        return client;
    }
}

```

Sử dụng đối tượng CloseableHttpClient để gọi webservice:

```

public static String sendMessageToAdapter(BusinessAdapterDTO baseMessage) throws
Exception {

    String jsonString = MappingObjectJson.convertObjectToJson(baseMessage);
    String username = "";
    if(baseMessage!=null && baseMessage.getSender_data()!=null){
        username = baseMessage.getSender_data().getUsername();
    }
    long startTime = System.currentTimeMillis();
    logger.info("username: "+username+", Input BusinessAdapter: "+jsonString);
    String url = getBusinessAdapterUrl() + "/api/business_adapter/runService";
    String result = null;
    HttpPost post = new HttpPost(url);
    StringEntity params = new StringEntity(jsonString, ContentType.APPLICATION_JSON);
    post.addHeader("content-type", "application/json; charset=utf-8");
    post.setEntity(params);
    post.setHeader("USERNAME", getUsername());
    post.setHeader("PASSWORD", getPassword());
    try {CloseableHttpClient httpClient = HttpClientFactory.getHttpClientFromPool();
        CloseableHttpResponse httpResponse = httpClient.execute(post) }
        result = EntityUtils.toString(httpResponse.getEntity());
        long endTime = System.currentTimeMillis();
        logger.info("username: "+username+", finished call BusinessAdapter:" +
result+", process time: "+String.valueOf(endTime - startTime) + " millisecond");
    }
    return result;
}
}

```

#### 42. Đảm bảo hiệu năng thao tác cập nhật dữ liệu lên redis sử dụng thư viện jedis

Khi cập nhật dữ liệu vào redis tránh gọi lệnh sync (ví dụ sử dụng jedis.sync trong thư viện jedis) đặc biệt là các trường hợp hệ thống ghi các đối tượng dữ liệu lớn khi sử dụng lệnh này sẽ dẫn đến node master của redis thực hiện ngay việc đồng bộ dữ liệu sang các node slave gây chậm cụm redis.

Ví dụ đoạn code sai (đúng cần bỏ dòng `jedis.sync()`)

```

public static void saveUserToServer(UserData userData, String server) {

```

```

Jedis jedis = null;
try {
    logger.info("====> Start saveUserToServer");
    jedis = JedisSentinelConnectionPool.getRedis();
    jedis.select(Constants.REDIS.DB_DEFAULT);
    String now = Util.getSysdate("yyyyMMddHHmmss");
    jedis.hset(Constants.REDIS.SERVER_GROUP + userData.getUsername(), server,
now);
    logger.info("====> End saveUserToServer success");
    jedis.sync();
} catch (Exception e) {
    logger.error("====> saveUserToServer have error: ", e);
} finally {
    JedisSentinelConnectionPool.closeJedis(jedis);
}
}

```

#### 43. Lưu ý khi sử dụng synchronized

Phần này bổ sung thêm một số kinh nghiệm về sử dụng synchronized ngoài nội dung trong mục 2. Không sử dụng "Lock" trong khối synchronized

Trong đó đưa ra các ví dụ để hạn chế việc sử dụng synchronized

+ Hàm gọi webservice của hệ thống khác không để synchronized

Ví dụ đoạn code không tuân thủ (đúng cần bỏ **synchronized**)

```

public static synchronized BaseChatbotResponse sendMessageToBot(String waAddress, JSONObject
baseMessage) {
    String jsonString = baseMessage.toString();
    String url = waAddress + "/receiveQuestion";
    BaseChatbotResponse result = null;
    Map<String, String> headers = new HashMap<>();
    try {
        String result2 = sendMessageToRestapi(url, jsonString, headers);
        if (Constants.MESSAGE_TIMEOUT_MESSAGE.equalsIgnoreCase(result2)) {
            String token =
MemoryDataLoaderRedis.getTokenByUsername(baseMessage.getJSONObject("userData").getString("username"));
OkHttpClient caller = new OkHttpClient.Builder().addInterceptor(new Interceptor() {
@Override public Response intercept(Chain chain) throws IOException {
return chain.proceed(chain.request().newBuilder().header("Authorization", token).build());
}
}).build();
result = gson.fromJson(result2, BaseChatbotResponse.class);
logger.info("Ket qua sau parser:" + result);
return result;
} catch (Exception ex) {
logger.error(ex.getMessage(), ex);
}
return result;
}

```



+ Khi code có cập nhật giá trị vào một biến/đối tượng cần xem xét sử dụng các đối tượng hỗ trợ multi thread concurrency.

- Ví dụ trường hợp hệ thống cần lưu một biến để quản lý số lượng giao dịch đồng thời vào một webservice, cần viết một class WebserviceUtils trong đó có một biến **totalCurrentRequest** để đếm số lượt kết nối. Khi bắt đầu giao dịch thì sẽ chạy vào hàm **increase** và tăng biến **totalCurrentRequest** 1 đơn vị. Khi kết thúc giao dịch sẽ chạy vào hàm **releaseRequest** để giảm giá trị biến **totalCurrentRequest** 1 đơn vị, như vậy **totalCurrentRequest** là số lượng giao dịch đồng thời.
- Nếu sử dụng synchronized

```
import java.util.concurrent.atomic.AtomicInteger;
import org.apache.log4j.Logger;

public class WebserviceUtils {

    private Logger logger;
    private Integer totalCurrentRequest = 0;
    private Long maxConnection;
    private String wsCode;

    public WebserviceUtils(Long maxConnection, Logger logger) {
        this.maxConnection = maxConnection;
        this.logger = logger;
    }

    /**
     * Tăng request lên
     */
    public synchronized void increase() {
        totalCurrentRequest++;
    }

    /**
     * Giải phóng request
     */
    public synchronized void releaseRequest() {
        if (totalCurrentRequest > 0) {
            totalCurrentRequest--;
        }
    }
}
```

Code sử dụng AtomicInteger cho tốc độ xử lý nhanh hơn khi có nhiều giao dịch:

```
import java.util.concurrent.atomic.AtomicInteger;
import org.apache.log4j.Logger;

public class WebserviceUtils {
```

```
private Logger logger;  
private AtomicInteger totalCurrentRequest = new AtomicInteger(0);  
private Long maxConnection;  
private String wsCode;
```

```
public WebserviceUtils(Long maxConnection, Logger logger) {  
    this.maxConnection = maxConnection;  
    this.logger = logger;  
}
```

```
/**  
 * Tang request len 1  
 */
```

```
public void increase() {  
    totalCurrentRequest.incrementAndGet();  
}
```

```
/**  
 * Giai phong request  
 */
```

```
public void releaseRequest() {  
    if (totalCurrentRequest.get() > 0) {  
        totalCurrentRequest.decrementAndGet();  
    }  
}
```

```
}
```



HTML, CSS

STT	Tên yêu cầu
1	Không dùng "NaN" trong phép toán so sánh
2	Sử dụng phép toán logic (short-circuit) để tránh lỗi truy cập vào vùng nhớ null trong các điều kiện so sánh
3	Không sử dụng toán tử "delete" với kiểu Array
4	Không thao tác dữ liệu với các biến "eval" và "arguments"
5	Biến tăng trong vòng "for" phải là biến trong điều kiện dừng vòng lặp
6	Kiểm soát khi truyền thông điệp qua văn bản
7	Xóa bỏ lệnh "debugger" trước khi triển khai
8	Không sử dụng các biểu thức hiển nhiên luôn đúng hoặc luôn sai
9	Không sử dụng trùng tên thuộc tính khi khai báo một đối tượng
10	Điều kiện trong các khối lệnh "if/else if" hoặc "switch...case" không trùng nhau
11	Sử dụng thuộc tính "length" để xác định các DOM elements
12	Không sử dụng biến "this" toàn cục
13	Tên các thuộc tính model không có dấu cách " "
14	Toán tử "+" và "-" không sử dụng với Object
15	Trong thẻ input "password" đặt thuộc tính "autocomplete" là "off"



## 1. Không dùng "NaN" trong phép toán so sánh

Giá trị NaN không bằng bất kỳ giá trị nào khác ngay cả chính nó. Thay vào đó nên sử dụng hàm isNaN() để so sánh giá trị của một biến có phải là kiểu số hay không.

```
- isNaN(a) = false => a là kiểu số  
- isNaN(a) = true => a không phải kiểu số
```

```
isNaN(123) //false  
isNaN(-1.23) //false  
isNaN(5-2) //false  
isNaN(0) //false  
isNaN('123') //false  
isNaN('Hello') //true  
isNaN('2005/12/12') //true  
isNaN('') //false  
isNaN(true) //false  
isNaN(undefined) //true  
isNaN('NaN') //true  
isNaN(NaN) //true  
isNaN(0 / 0) //true
```

Ví dụ code không tuân thủ:

```
var a = NaN;  
  
if (a === NaN) { // luôn trả về false  
  console.log("a is not a number");  
}  
if (a !== NaN) { // luôn trả về true  
  console.log("a is not NaN");  
}
```

Cách viết đúng:

```
if ( isNaN(a) ) {  
  console.log("a is not a number");  
} else {  
  console.log("a is not NaN");  
}
```

## 2. Sử dụng phép toán logic (short-circuit) để tránh lỗi truy cập vào vùng nhớ null trong các điều kiện so sánh.

Khi một điều kiện kiểm tra kết quả null, nếu kiểm tra tiếp các điều kiện còn lại có thể dẫn đến lỗi TypeError.

Ví dụ code không tuân thủ

```

if (str == null && str.length == 0) {
    console.log("String is empty");
}

if (str == undefined && str.length == 0) {
    console.log("String is empty");
}

if (str != null || str.length > 0) {
    console.log("String is not empty");
}

if (str != undefined || str.length > 0) {
    console.log("String is not empty");
}

```

Cách viết đúng:

```

if (str != null && str.length == 0) {
    console.log("String is empty");
}

if (str != undefined && str.length == 0) {
    console.log("String is empty");
}

if (str == null || str.length > 0) {
    console.log("String is not empty");
}

if (str == undefined || str.length > 0) {
    console.log("String is not empty");
}

```

### 3. Không sử dụng toán tử "delete" với kiểu Array

Toán tử delete được sử dụng để xóa một thuộc tính khỏi đối tượng. Trường hợp đối tượng là kiểu Array, toán tử delete cũng được sử dụng tương tự, nhưng nếu dùng toán tử này thì chỉ số của các phần tử phía sau phần tử bị xóa không nhảy lên chỉ số của vị trí phía trước.

Để xóa một phần tử trong mảng và các phần tử phía sau dồn lên phía trước lấp vào vị trí phần tử bị xóa thì nên dùng các hàm sau:

Array.prototype.splice - thêm/xóa phần tử trong mảng

Array.prototype.pop - thêm/xóa phần tử phía cuối của mảng

Array.prototype.shift - thêm/xóa phần tử phía đầu của mảng

Ví dụ code không tuân thủ

```
var myArray = ['a', 'b', 'c', 'd'];

delete myArray[2]; // Kết quả mảng: myArray => ['a', 'b', undefined, 'd']
console.log(myArray[2]); // Kết quả in ra: undefined
```

Cách viết đúng:

```
var myArray = ['a', 'b', 'c', 'd'];

// Xóa phần tử ở vị trí chỉ số = 2
removed = myArray.splice(2, 1); // Kết quả mảng: myArray => ['a', 'b', 'd']
console.log(myArray[2]); // Kết quả in ra: 'd'
```

#### 4. Không thao tác dữ liệu với các biến "eval" và "arguments".

Trong Javascript, hàm eval() được sử dụng để tính toán các giá trị. Arguments được sử dụng để truy cập các tham số thông qua chỉ số index. Việc thao tác dữ liệu với các đối tượng này có thể dẫn đến phát sinh lỗi ngoài mong muốn.

Ví dụ code không tuân thủ

```
eval = 17; // Noncompliant
arguments++; // Noncompliant
++eval; // Noncompliant
var obj = { set p(arguments) { } }; // Noncompliant
var eval; // Noncompliant
try { } catch (arguments) { } // Noncompliant
function x(eval) { } // Noncompliant
function arguments() { } // Noncompliant
var y = function eval() { }; // Noncompliant
var f = new Function("arguments", "return 17;"); // Noncompliant

function fun() {
  if (arguments.length == 0) { // Compliant
    // do something
  }
}
```

Cách viết đúng:

```
result = 17;
args++;
++result;
var obj = { set p(arg) { } };
var result;
try { } catch (args) { }
function x(arg) { }
function args() { }
var y = function fun() { };
var f = new Function("args", "return 17;");

function fun() {
  if (arguments.length == 0) {
```



```
// do something
}
}
```

## 5. Biến tăng trong vòng "for" phải là biến trong điều kiện dừng vòng lặp

Khi biến tăng và biến điều kiện dừng vòng lặp for không giống nhau thì thường là lỗi có thể dẫn đến vòng lặp không bao giờ kết thúc và nếu như không gây lỗi thì cũng rất khó cho việc bảo trì về sau.

Ví dụ code không tuân thủ

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

Cách viết đúng:

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

## 6. Kiểm soát khi truyền thông điệp qua văn bản

HTML5 cho phép gửi thông điệp từ một trang HTML tới một trang HTML ở một địa chỉ domain khác. Để tránh nguy cơ lộ thông tin nhạy cảm khi gửi tới một domain không an toàn thì dữ liệu khi gửi đi trong hàm postMessage() cần được kiểm duyệt trước khi gửi.

Ví dụ code không tuân thủ

```
var myWindow = document.getElementById('myIFrame').contentWindow;
myWindow.postMessage(message, ""); // Dữ liệu trong 'myIFrame' trước khi
gửi đi có thể chứa thông tin nhạy cảm?
```

## 7. Xóa bỏ lệnh "debugger" trước khi triển khai

"debugger" là lệnh do lập trình viên sử dụng để tìm lỗi trong quá trình phát triển ứng dụng. Sau khi đóng gói sản phẩm trong giai đoạn triển khai tất cả câu lệnh 'debugger' cần được xóa khỏi source code.

Ví dụ code không tuân thủ

```
for (i = 1; i < 5; i++) {
    Debug.write("loop index is " + i);
    debugger;
}
```

Cách viết đúng:

```
for (i = 1; i < 5; i++) {
```



```
Debug.write("loop index is " + i);  
}
```

## 8. Không sử dụng các biểu thức hiển nhiên luôn đúng hoặc luôn sai

Xảy ra trường hợp này thường là do nhầm lẫn (copy/paste) hoặc đơn giản là code thừa, gây khó khăn cho việc bảo trì.

Ví dụ code không tuân thủ

```
if ( a == a ) { // luôn đúng  
doZ();  
}  
if ( a != a ) { // luôn sai  
doY();  
}  
if ( a == b && a == b ) {  
doX();  
}  
if ( a == b || a == b ) {  
doW();  
}  
  
var j = 5 / 5; //luôn = 1  
var k = 5 - 5; //luôn = 0
```

Cách viết đúng:

```
doZ();  
  
if ( a == b ) {  
doX();  
}  
if ( a == b ) {  
doW();  
}  
  
var j = 1;  
var k = 0;
```

## 9. Không sử dụng trùng tên thuộc tính khi khai báo một đối tượng

Javascript chấp nhận khai báo trùng thuộc tính nhưng khi xuất hiện nhiều thuộc tính có tên trùng nhau thì Javascript sẽ chỉ cập nhật giá trị thuộc tính khai báo sau cùng và bỏ qua thuộc tính trùng đã khai báo trước đó

Ví dụ code không tuân thủ

```
var data = {  
  "key": "value",  
  "i": "value",  
  "key": "value", // Noncompliant - duplicate of "key"  
}
```

```
'key': "value", // Noncompliant - duplicate of "key"
key: "value", // Noncompliant - duplicate of "key"
\u006bey: "value", // Noncompliant - duplicate of "key"
"\u006bey": "value", // Noncompliant - duplicate of "key"
"\x6bey": "value", // Noncompliant - duplicate of "key"
1: "value" // Noncompliant - duplicate of "1"
}
```

Cách viết đúng:

```
var data = {
  "key": "value",
  "1": "value",
  "key2": "value",
  'key3': "value",
  key4: "value",
  \u006bey5: "value",
  "\u006bey6": "value",
  "\x6bey7": "value",
  1b: "value"
}
```

#### 10. Điều kiện trong các khối lệnh "if/else if" hoặc "switch...case" không trùng nhau.

Trong các khối lệnh "if/else if" hoặc "switch...case", chỉ một nhánh đầu tiên có điều kiện đúng được thực hiện. Do đó việc lặp lại một điều kiện (thường do nhầm lẫn khi copy/paste) có thể gây ra các lỗi nghiêm trọng mà chúng ta không lường trước được.

Ví dụ code không tuân thủ

```
if (param == 1)
  openWindow();
else if (param == 2)
  closeWindow();
else if (param == 1) // Đã có điều kiện này
  moveWindowToTheBackground();

switch(i) {
  case 1:
    //...
    break;
  case 3:
    //...
    break;
  case 1: // Đã có case này
    //...
    break;
  default:
    //...
    break;
}
```

Cách viết đúng:

```

if (param == 1)
    openWindow();
else if (param == 2)
    closeWindow();
else if (param == 3)
    moveWindowToTheBackground();

switch(i) {
    case 1:
        //...
        break;
    case 3:
        //...
        break;
    default:
        // ...
        break;
}

```

## 11. Sử dụng thuộc tính "length" để xác định các DOM elements

Khi thực hiện tìm kiếm một đối tượng nên sử dụng thuộc tính length để xác định có tìm thấy đối tượng hay không.

Ví dụ code không tuân thủ

```

if ( $( "div.foo" ) ) { // Luôn luôn trả về true mặc dù không tìm thấy đối tượng nào
}

```

Cách viết đúng:

```

// Testing whether a selection contains elements.
if ( $( "div.foo" ).length > 0 ) {
    // this code only runs if elements were found
    // ...
}

```

## 12. Không sử dụng biến "this" toàn cục

Khi sử dụng biến "this" toàn cục khai báo ngoài cùng thì javascript sẽ hiểu là tham chiếu tới đối tượng window. Nếu muốn khai báo biến toàn cục thì loại bỏ biến this vẫn cho một kết quả tương tự.

Ví dụ code không tuân thủ

```

this.foo = 1; // Không tuân thủ
console.log(this.foo); // Không tuân thủ

function MyObj() {
    this.foo = 1; // Tuân thủ
}

```

```
MyObj.func1 = function() {
  if (this.foo == 1) { // Tuân thủ
    // ...
  }
}
```

Cách viết đúng:

```
foo = 1;
console.log(foo);

function MyObj() {
  this.foo = 1;
}

MyObj.func1 = function() {
  if (this.foo == 1) {
    // ...
  }
}
```

### 13. Tên các thuộc tính model không có dấu cách " "

Khi sử dụng framework Backbone.js, tên các thuộc tính model không nên chứa dấu cách vì đối tượng Events chấp nhận danh sách event được xác định bằng dấu cách. Do đó tên một thuộc tính chứa dấu cách có thể bị hiểu sai ý nghĩa.

Ví dụ code không tuân thủ

```
Person = Backbone.Model.extend({
  defaults: {
    'first name': 'Bob', // Không tuân thủ
    'birth date': new Date() // Không tuân thủ
  },
});
```

Cách viết đúng:

```
Person = Backbone.Model.extend({
  defaults: {
    firstName: 'Bob',
    birthDate: new Date()
  },
});
```

### 14. Toán tử "+" và "-" không sử dụng với Object

Toán tử + và - được sử dụng để chuyển kiểu giá trị sang giá trị số, tuy nhiên không phải mọi giá trị để có thể chuyển sang Number, khi đó kết quả luôn trả về NaN.

Ví dụ code không tuân thủ



```
var obj = {x : 1};
doSomethingWithNumber(+obj); // Không theo luật

function foo(){
  return 1;
}
doSomethingWithNumber(-foo); //Không theo luật
```

Cách viết đúng:

```
var obj = {x : 1};
doSomethingWithNumber(+obj.x);

function foo(){
  return 1;
}
doSomethingWithNumber(-foo());

var str = '42';
doSomethingWithNumber(+str);

//Ngoại lệ
//Toán tử +, - có thể dùng với đối tượng thuộc kiểu primitive.

var b = new Boolean(true);
doSomethingWithNumber(-b); // Compliant
```

#### 15. Trong thẻ input "password" đặt thuộc tính "autocomplete" là "off"

Hầu hết các trình duyệt tự động điền nội dung thẻ input 'password' khi password đã được nhập trước đó. Điều này có thể gây ra lỗi mất an toàn thông tin. Trong HTML có thể khắc phục bằng cách đặt thuộc tính autocomplete cho thẻ này là off.

Ví dụ code không tuân thủ

```
HTML5:
<input type="password" />
```

Cách viết đúng:

```
HTML5:
<input type="password" autocomplete="off" />
```