PartII(a)

1. CID 10065

Missing break in switch (MISSING\_BREAK)

Classification:False Positive

Explanation:

In “case 3”( BooleanUtils.java from line 628 to line 640), if “ ch == ‘y’/’Y’ ”,it will return the corresponding Boolean vaule after checking whether it equals to “yes” or “YES”.Due to the lack of “break”,it will execution “case4"if it does not satisfy any case in “case 3”,and it will also does not any case of “case 4” since the unequal length.Finally,it will return false,which does not modify the output.

1. CID 10066

CN:Bad implementation of cloneable idiom(FB.CN-IDIOM)

Classification:False Positive

Explanation:

In this class, it implements Cloneable interface,but have clone() method is not called in it.It will waste the space.However,it will not cause the problem.

1. CID 10067

Dm: Dubious method used (FB.DM\_DEFAULT\_ENCODING)

Classification:Bug

Faulty lines:(NestableDelegate.java line292)

PrintWriter pw = new PrintWriter(out, false);

Bug fix:

PrintWriter pw = new PrintWriter((new Printstream(out),”UTF-8”), false);

1. CID 10068

Dm: Dubious method used (FB.DM\_NEXTINT\_VIA\_NEXTDOUBLE)

Classification:Intentional

Original code:

110        **return** (int)(Math.random() \* n);

Re-factory:

110        **return** (int)( Random.nextInt(n));

Explanation:

Math.random() requires about twice the processing to generate an integer and is subject to synchronization. Random.nextInt(n) uses Random.next() less than twice on average- it uses it once, and if the value obtained is above the highest multiple of n below MAX\_INT it tries again, otherwise is returns the value modulo n (this prevents the values above the highest multiple of n below MAX\_INT skewing the distribution), so returning a value which is uniformly distributed in the range 0 to n-1.Therefore Random.nextInt(n) is more efficient and less biased.

1. CID 10069

Eq: Problems with implementation of equals() (FB.EQ\_COMPARING\_CLASS\_NAMES)

Classification:Bug

Faulty lines:( Enum.java line 552)

**if** (other.getClass().getName().equals(**this**.getClass().getName()) == false)

Bug fix: (remove getClass())

**if** (other.getName().equals(**this**.getName()) == false)

1. CID 10070

Eq: Problems with implementation of equals() (FB.EQ\_COMPARING\_CLASS\_NAMES)

Classification:Bug

Faulty lines:( Enum.java line 598)

**if** (other.getClass().getName().equals(**this**.getClass().getName()) == false)

Bug fix: (remove getClass())

**if** (other.getName().equals(**this**.getName()) == false)

1. CID 10071

ES: Checking String equality using == or != (FB.ES\_COMPARING\_PARAMETER\_STRING\_WITH\_EQ)

Classification:Intentional

Original code:(BooleanUnits.java line 614)

614        **if** (str == "true")

Re-factory :

614        **if** (str.equals("true"))

Explanation:

According to the comments above the method and above this code line, the parameter of method toBoolean(String str) is interned strings, so using “==” to compare str with “true” would not cause problem.

But, to achieve scalability, if the parameter str is not a String constant or interned string, the result of this statement would not be functionally right. Using “equals” is a better way to promise the program function.

1. CID 10072

ES: Checking String equality using == or != (FB.ES\_COMPARING\_PARAMETER\_STRING\_WITH\_EQ)

Classification:Intentional

Original code :(StringUtils.java line 4865)

4865        **if** (str1 == str2)

Re-factory :

4865        **if** (str1.equals("str2"))

Explanation:

According to the comments above the method and above this code line, the parameter of method indexOfDifference(String str1, String str2)

is interned strings, so using “==” to compare str1 with “str2” would not cause problem.

But to achieve scalability, if the parameter str1 or str2 is not a String constant or interned string, the result of this statement would not be functionally right. Using “equals” is a better way to promise the program function.

1. CID 10073

ES: Checking String equality using == or != (FB.ES\_COMPARING\_PARAMETER\_STRINGS\_WITH\_EQ)

Classification: False Positive

Original code: In DurationFormatUnits.java

1. 409            **else** **if** (value == S)
2. 405            **else** **if** (value == s)
3. 401 **else** **if** (value == m)
4. 397 **else** **if** (value == H)
5. 393 **else** **if** (value == d)
6. 389 **else** **if** (value == M)
7. 385 **else** **if** (value == y)

Explanation:

380            Object value = token.getValue();

“value” is a constant which is defined in line 380. And for

else if statement,both the right hand and the left hand

are constant strings,so they can be compared by “==”.

CID 10074

ES: Checking String equality using == or != (FB.ES\_COMPARING\_PARAMETER\_STRING\_WITH\_EQ)

Classification:False Positive

Explanation:

previous.getValue() is constant in the source file, variable value just be assigned to a String value, but not instantiate a new object. So they can be compared using “==”.

CID 10075

IM: Questionable integer math (FB.IM\_AVERAGE\_COMPUTATION\_COULD\_OVERFLOW)

Classification:Intentional

Original code : In Entities.java(line 649)

649                int mid = (low + high) >> 1;

Re-factory:

649                int mid = (low + high) >>> 1;

Explanation:

If the result of (low + high) is zero or positive,it will not cause the problem.While the result is negative, “>>” will cause the wrong answer.

The >> operator shifts a 1 bit into the most significant bit if it was a 1, and the >>> shifts in a 0 regardless.

(low + high) >> 1 keeps the sign bit of the original, so a negative value for a gives a negative result.

(low + high)  >>> 1 works by introducing a zero sign bit, so the result cannot be negative for any (low + high).

CID 10076

NP: Null pointer dereference (FB.NP\_BOOLEAN\_RETURN\_NULL)

Classification:Intentional

Solution:left as-is

Explanation:

CID 10077

NP: Null pointer dereference (FB.NP\_BOOLEAN\_RETURN\_NULL)

Classification:Intentional

Solution:left as-is

CID 10078

NP: Null pointer dereference (FB.NP\_BOOLEAN\_RETURN\_NULL)

Classification:Intentional

Solution:left as-is

CID 10079

NP: Null pointer dereference (FB.NP\_BOOLEAN\_RETURN\_NULL)

Classification:Intentional

Solution:left as-is

CID 10080

NP: Null pointer dereference (FB.NP\_BOOLEAN\_RETURN\_NULL)

Classification:Intentional

Solution:left as-is

CID 10081

NP: Null pointer dereference (FB.NP\_BOOLEAN\_RETURN\_NULL)

Classification:Intentional

Solution:left as-is

CID 10082

REC: RuntimeException capture (FB.REC\_CATCH\_EXCEPTION)

Classification:Intentional

Original code :In ExceptionUnits.java (line 97)

 97        **catch** (Exception e)

Re-factory:

 97        **catch** (RuntimeException e)

Explanation:

In Java, there are two types of exceptions: checked exceptions and un-checked exceptions. A checked exception must be handled explicitly by the code, whereas, an un-checked exception does not need to be explicitly handled.

Any exception that derives from "Exception" is a checked exception.

Any exception that derives from "RuntimeException" is an un-checked exceptions.

Generally,throwing a checked exception will not cause the problem.While the method could not be normally handle an un-checked exceptions .  RuntimeExceptions do not need to be explicitly handled by the calling code.

CID 10083

Se: Incorrect definition of Serializable class (FB.SE\_BAD\_FIELD)

Classification: Intentional

Original code:In FastDateFormat.java (line 137)

137    **private** Rule[] mRules;

Re-factory:

137    **private transient** Rule[] mRules;

Explanation:

By default, all of object's variables get converted into a persistent state. In some cases, you may want to avoid persisting some variables because you don't have the need to persist those variables. So you can declare those variables as transient. If the variable is declared as transient, then it will not be persisted.

CID 10084

UrF: Unread field (FB.URF\_UNREAD\_FIELD)

Classification:False Positive

Explanation:

 85            **this**.key = key;

the key has never been read,and it seems always same with hash and is redundant .But it will not cause the problem.