Notification 1 (string comparison using == or !=):

This code compares java.lang.String objects for reference equality using the == or != operators. Unless both strings are either constants in a source file, or have been interned using the String.intern() method, the same string value may be represented by two different String objects. Consider using the equals(Object) method instead.

You shouldn’t compare strings using == or != because it is only comparing the reference not the actual string itself. Comparing strings is done using the .equals() method.

Notification 2 (incorrect lazy initialization):

This method contains an unsynchronized lazy initialization of a static field. After the field is set, the object stored into that location is further updated or accessed. The setting of the field is visible to other threads as soon as it is set. If the further accesses in the method that set the field serve to initialize the object, then you
have a very serious multithreading bug, unless something else prevents any other thread from accessing the stored object until it is fully initialized.

Even if you feel confident that the method is never called by multiple threads, it might be better to not set the static field until the value you are setting it to is fully populated/initialized.

You are initializing a static variable without a synchronizing it, which if you trying to do lazy initialization is incorrect as the way the code is written now more than one of these objects can be created.

Notification 3 (Synchronize on a mutable field):

This method synchronizes on an object referenced from a mutable field. This is unlikely to have useful semantics, since different threads may be synchronizing on different objects.

It is possible for more than one of these objects to have been created or changed before it is actually synchronized on so there is no telling what it is in fact being synchronized.

Notification 4 (Redundant null check):

A value is checked here to see whether it is null, but this value can't be null
because it was previously dereferenced and if it were null a null pointer exception would have occurred at the earlier dereference. Essentially, this code and the previous dereference disagree as to whether this value is allowed to be null. Either the check is redundant or the previous dereference is erroneous.

The null check you are doing is not needed or misplaced. If e was null the code would break before reaching the null check. You should consider removing the null check and handing potential exception when e is dereferenced.

Notification 5 (Possible null pointer dereference):

```java
private int findLine(int offset) {
    int[] lineEnds = lineCache.get();
    if (offset < lineEnds[0]) {
        return 0;
    } else if (offset > lineEnds[lineCount - 1]) {
        return lineCount;
    } else {
        return findLine(lineEnds, offset, 0, lineCount - 1);
    }
}
```

The return value from a method is dereferenced without a null check, and the return value of that method is one that should generally be checked for null. This may lead to a `NullPointerException` when the code is executed.

You are trying to access data that may not exist. You should check lineEnds[0] for null before trying to access it.

Notification 6 (Unused code):

```
private org.omg.CORBA.ORB orb;
private Vector<String> _contexts;
```

You are not using (or reading from) this variable anywhere in this class (it’s a private variable so it’s not being used outside this class either). You could remove it to get rid of the error and the code would work the same.

Notification 7 (Parameterized/Raw type):
You created a generic object Vector<String> but did not properly initialize it. The new Vector should be new Vector<String>.

Notification 8 (unimplemented methods):

You are implementing a class (DirectBuffer) but not implementing all the required methods (viewedBuffer). If you implement this method the error will go away.

Notification 9 (serializable class needs serial ID):

Somewhere down the line of classes/interfaces being implemented/extended from this class, Serializable is being implemented. Proper usage of this interface requires a serialversionUID during deserialization to ensure that the classes loaded are compatible with respect to serialization.
Notification 10 (unimplemented methods):

```java
interruptor = new Interruptible()
public void interrupt(Thread target)
}{
  synchronized (closeLock)
  if (!open)
    return;
    open = false;
    interrupted = target;
}
```

There are methods from the interface you are trying to instantiated as an anonymous class that you are not implementing. You should implement an interrupt method with no parameters or change the method signature in the interface.

Notification 11 (method not applicable for arguments):

```java
if (me.isInterrupted())
  interruptor.interrupt(me);
```

You are trying to call an interrupt method that is not expected for this Interruptible object. You should either call the method you implemented with no parameters or make sure the method in the interface matches this one.
Notification 12 (Red class with red header):

```java
public class PlotUtilities {
    /**
     * Returns <code>true</code> if all the datasets belonging to the
     * plot are empty or <code>null</code>, and <code>false</code> otherwise.
     * @param plot the plot <code>null</code> permitted.
     * @return A boolean.
     * @since 1.0.7
     */
    public static boolean isEmptyOrNull(XYPlot plot) {
        if (plot != null) {
            for (int i = 0, n = plot.getDatasetCount(); i < n; i++) {
                final XYDataset dataset = plot.getDataset(i);
                if (!DatasetUtilities.isEmptyOrNull(dataset)) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

You have not instantiated an instance of this class (default constructor) nor have you called any of the methods.

Notification 13 (Red class--constructor only):

```java
public class XYCrossHairState extends CrosshairState {
    /**
     * Creates a new instance.
     */
    // TODO ECL Class 2
    public XYCrossHairState() {
    }
}
```

You have not created/instantiated an instance of this class (implemented constructor)

Notification 14 (Simple if statement 1 of 2 branches):

```java
public NormalDistributionFunction2D(double mean, double std) {
    if (std <= 0) {
        throw new IllegalArgumentException("Requires 'std' > 0.");
    }
}
```
You are only executing one branch of this 2 branch if statement (the false branch). You should run the method with input(s) that will execute the true branch of the if.

**Notification 15 (Short circuit return statement):**

In the case of the notifications they look at, the methods are not being called. However, each return statement mentions branches; 2 branches means you need to test pass = that number and also pass != to that number. 4 branches may mean when each part of the return statement returns true and false.

**Notification 16 (try/catch -- no exception caught):**

The try block has executed and no exception was caught so the catch block did not execute.

**Notification 17 (try/finally -- exception thrown, partial coverage in finally):**
The try attempted to execute but failed, which led to the finally being executed and then exiting the method. Because only failure of the try and execution of the finally was tested, the inside of the finally is yellow. If this same test were called twice, once with an exception and once without, presumably at least the inside of the finally would be green. The red bracket at the end of the method suggests that the method exited after executing the finally.

Notification 18 (try/finally -- try executed, partial coverage in finally):

This is the opposite of N17. The try did execute which means the finally does not. Because this code was only called once (with no exception) the inside of the finally is yellow.

Notification 19 (try/catch -- exception caught):

The try attempted to executed but failed; an exception was thrown and caught so the catch block was executed.
Notification 20 (try/finally -- method exits):

```java
public String getURL(int series, int item) {
    String result = null;
    if (series < getListCount()) {
        List urls = (List) this.urlSeries.get(series);
        if (urls != null) {
            if (item < urls.size()) {
                result = (String) urls.get(item);
            }
        }
    } else {
        return result;
    }
}
```

This is similar to N17 except here there is more code in the method instead of just the closing bracket so we can see more clearly that the method exited once the finally executed.

Notification 21 (Nested if statements):

```java
try {
    TimeZone zone = TimeZone.getTimeZone("America/Los_Angeles");
    assertEquals(-2991360000L, w.getLastMillisecond(zone));
} finally {
    Locale.setDefault(saved);
}
```

// try null zone
try {
    w.getLastMillisecond((TimeZone) null);
} catch (NullPointerException e) {
    pass = true;
}
assertFalse(pass);
```
Only the true branches of each conditional executed (the only path that does anything).