Non-serializable field

Name: Non-serializable field

Description: A non-transient field in a serializable class must also be serializable otherwise it causes the class to fail to serialize with a 'NotSerializableException'.

ID: java/non-serializable-field

Kind: problem

Severity: warning

Precision: low

If a serializable class is serialized using the default Java serialization mechanism, each non-static, non-transient field in the class must also be serializable. Otherwise, the class generates a java.io.NotSerializableException as its fields are written out by ObjectOutputStream.writeObject.

As an exception, classes that define their own readObject and writeObject methods can have fields that are not themselves serializable. The readObject and writeObject methods are responsible for encoding any state in those fields that needs to be serialized.

Recommendation

To avoid causing a NotSerializableException, do one of the following:

- **Mark the field as transient**: Marking the field as transient makes the serialization mechanism skip the field. Before doing this, make sure that the field is not really intended to be part of the persistent state of the object.
- **Define custom readObject and writeObject methods for the Serializable class**: Explicitly defining the readObject and writeObject methods enables you to choose which fields to read from, or write to, an object stream during serialization.
- **Make the type of the field Serializable**: If the field is part of the object's persistent state and you wish to use Java's default serialization mechanism, the type of the field must implement Serializable. When choosing this option, make sure that you follow best practices for serialization.

Example 1

In the following example, WrongPerformanceRecord contains a field factors that is not serializable but is in a serializable class. This causes a java.io.NotSerializableException when the field is written out by writeObject. However, PerformanceRecord contains a field factors that is marked as transient, so that the serialization mechanism skips the field. This means that a correctly serialized record is output by writeObject.

```java
class DerivedFactors {
    private Number efficiency; // performance record
    private Number costPerItem;
    private Number profitPerItem;
    ...
}

class WrongPerformanceRecord implements Serializable {
    private String unitId;
    private Number dailyThroughput;
    private Number dailyCost;
    private DerivedFactors factors; // BAD: 'DerivedFactors' is not serializable
    // but is in a serializable class. This
```
// causes a 'java.io.NotSerializableException'
// when 'WrongPerformanceRecord' is serialized.
...

class PerformanceRecord implements Serializable {
  private String unitId;
  private Number dailyThroughput;
  private Number dailyCost;
  transient private DerivedFactors factors; // GOOD: 'DerivedFactors' is declared
  // 'transient' so it does not contribute to
  // serializable state of 'PerformanceRecord'.
...
}

Example 2

In this second example, WrongPair takes two generic parameters L and R. The class itself is serializable, but users of this class are not forced to pass serializable objects to its constructor, which could lead to problems during serialization. The solution is to set upper type bounds for the parameters, to force the user to supply only serializable objects. A similar example is the WrongEvent class, which takes a weakly typed eventData object. A better solution is to force the user to supply an object whose class implements the Serializable interface.

class WrongPair<L, R> implements Serializable{
  private final L left;          // BAD: L and R are not guaranteed to be serializable
  private final R right;
  public WrongPair(L left, R right){ ... }
...
}

class Pair<L extends Serializable, R extends Serializable> implements Serializable{
  private final L left;          // GOOD: L and R must implement Serializable
  private final R right;
  public Pair(L left, R right){ ... }
...
}

class WrongEvent implements Serializable{
  private Object eventData;       // BAD: Type is too general.
  public WrongEvent(Object eventData){ ... }
}

class Event implements Serializable{
  private Serializable eventData; // GOOD: Force the user to supply only serializable data
  public Event(Serializable eventData){ ... }
}

References

- Java API Documentation: Serializable, ObjectOutputStream.