MET11-J. Ensure that keys used in comparison operations are immutable

Objects that serve as keys in ordered sets and maps should be immutable. When some fields must be mutable, the `equals()`, `hashCode()`, and `compareTo()` methods must consider only immutable state when comparing objects. Violations of this rule can produce inconsistent orderings in collections. The documentation of `java.util.Interface Set<E>` and `java.util.Interface Map<K,V>` warns against this. For example, the documentation for the Interface `Map` states [API 2014]:

> *Note: great care must be exercised [when] mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. While it is permissible for a map to contain itself as a value, extreme caution is advised: the `equals` and `hashCode` methods are no longer well defined on such a map.*

Noncompliant Code Example

This noncompliant code example defines a mutable class `Employee` that consists of the fields `name` and `salary`, whose values can be changed using the `setEmployeeName()` and `setSalary()` method. The `equals()` method is overridden to provide a comparison facility by employee name.

```
// Mutable class Employee
class Employee {
  private String name;
  private double salary;

  Employee(String empName, double empSalary) {
    this.name = empName;
    this.salary = empSalary;
  }

  public void setEmployeeName(String empName) {
    this.name = empName;
  }

  public void setSalary(double empSalary) {
    this.salary = empSalary;
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Employee)) {
      return false;
    }

    Employee emp = (Employee)o;
    return emp.name.equals(name);
  }

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

}

// Client code
Map<Employee, Calendar> map =
  new ConcurrentHashMap<Employee, Calendar>();
// ...
```

Use of the `Employee` object as a key to the map is insecure because the properties of the object could change after an ordering has been established. For example, a client could modify the `name` field when the last name of an employee changes. As a result, clients would observe nondeterministic behavior.

Compliant Solution

This compliant solution adds a final field `employeeID` that is immutable after initialization. The `equals()` method compares `Employee` objects on the basis of this field.

```
// Mutable class Employee
class Employee {
  private String name;
  private double salary;
  private final long employeeID;  // Unique for each Employee

  Employee(String name, double salary, long empID) {
    this.name = name;
    this.salary = salary;
    this.employeeID = empID;
  }

  // ...

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Employee)) {
      return false;
    }

    Employee emp = (Employee)o;
    return emp.employeeID == employeeID;
  }
}

// Client code remains same
Map<Employee, Calendar> map =
  new ConcurrentHashMap<Employee, Calendar>();
// ...
```

The `Employee` class can now be safely used as a key for the map in the client code.

Noncompliant Code Example

Many programmers are surprised by an instance of hash code mutability that arises because of serialization. The contract for the `hashCode()` method lacks any requirement that hash codes remain consistent across different executions of an application. Similarly, when an object is serialized and subsequently deserialized, its hash code after deserialization may be inconsistent with its original hash code.

This noncompliant code example uses the `MyKey` class as the key index for the `Hashtable`. The `MyKey` class overrides `Object.equals()` but uses the default `Object.hashCode()`. According to the Java API [API 2014] class `Hashtable` documentation:

> *To successfully store and retrieve objects from a hash table, the objects used as keys must implement the `hashCode` method and the `equals` method.*

This noncompliant code example follows that advice but nevertheless can fail after serialization and deserialization. Consequently, it may be impossible to retrieve the value of the object after deserialization by using the original key.

```
class MyKey implements Serializable {
  // Does not override hashCode()
}

class HashSer {
  public static void main(String[] args)
                    throws IOException, ClassNotFoundException {
    Hashtable<MyKey,String> ht = new Hashtable<MyKey, String>();
    MyKey key = new MyKey();
    ht.put(key, "Value");
    System.out.println("Entry: " + ht.get(key));
    // Retrieve using the key, works

    // Serialize the Hashtable object
    FileOutputStream fos = new FileOutputStream("hashdata.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(ht);
    oos.close();

    // Deserialize the Hashtable object
    FileInputStream fis = new FileInputStream("hashdata.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Hashtable<MyKey, String> ht_in =
        (Hashtable<MyKey, String>)(ois.readObject());
    ois.close();

    if (ht_in.contains("Value"))
      // Check whether the object actually exists in the hash table
      System.out.println("Value was found in deserialized object.");

    if (ht_in.get(key) == null) // Gets printed
      System.out.println(
          "Object was not found when retrieved using the key.");
  }
}
```

Compliant Solution

This compliant solution changes the type of the key value to be an `Integer` object. Consequently, key values remain consistent across multiple runs of the program, across serialization and deserialization, and also across multiple Java Virtual Machines.

```
class HashSer {
  public static void main(String[] args)
                     throws IOException, ClassNotFoundException {
    Hashtable<Integer, String> ht = new Hashtable<Integer, String>();
    ht.put(new Integer(1), "Value");
    System.out.println("Entry: " + ht.get(1)); // Retrieve using the key

    // Serialize the Hashtable object
    FileOutputStream fos = new FileOutputStream("hashdata.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(ht);
    oos.close();

    // Deserialize the Hashtable object
    FileInputStream fis = new FileInputStream("hashdata.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    Hashtable<Integer, String> ht_in =
        (Hashtable<Integer, String>)(ois.readObject());
    ois.close();

    if (ht_in.contains("Value"))
      // Check whether the object actually exists in the Hashtable
      System.out.println("Value was found in deserialized object.");

    if (ht_in.get(1) == null)  // Not printed
      System.out.println(
          "Object was not found when retrieved using the key.");
  }
}
```

This problem could also have been avoided by overriding the `hashcode()` method in the `MyKey` class, though it is best to avoid serializing hash tables that are known to use implementation-defined parameters.

Risk Assessment

Failure to ensure that the keys used in a comparison operation are immutable can lead to nondeterministic behavior.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MET11-J	Low	Probable	High	**P2**	**L3**

Automated Detection

Some available static analysis tools can detect instances where the `compareTo()` method is reading from a nonconstant field.

Bibliography

[API 2014]	Class `Hashtable<K,V>`
	`java.util.Interface Set<E>`
	`java.util.Interface Map<K,V>`