OBJ01-J. Limit accessibility of fields

Invariants cannot be enforced for public nonfinal fields or for final fields that reference a mutable object. A protected member of an exported class represents a public commitment to an implementation detail. Attackers can manipulate such fields to violate class invariants, or they may be corrupted by multiple threads accessing them concurrently [Bloch 2008]. As a result, fields must be declared private or package-private.

Noncompliant Code Example (Public Primitive Field)

In this noncompliant code example, the `total` field tracks the total number of elements as they are added to and removed from a container using the methods `add()` and `remove()` respectively.

```
public class Widget {
  public int total; // Number of elements

  void add() {
    if (total < Integer.MAX_VALUE) {
      total++;
      // ...
    } else {
      throw new ArithmeticException("Overflow");
    }
  }

  void remove() {
    if (total > 0) {
      total--;
      // ...
    } else {
      throw new ArithmeticException("Overflow");
    }
  }
}
```

As a public field, `total` can be altered by client code independently of the `add()` and `remove()` methods.

Compliant Solution (Private Primitive Field)

Accessor methods provide controlled access to fields outside of the package in which their class is declared. This compliant solution declares `total` as private and provides a public accessor. The `add()` and `remove()` methods modify its value while preserving class invariants.

```
public class Widget {
  private int total; // Declared private

  public int getTotal () {
    return total;
  }

  // Definitions for add() and remove() remain the same
}
```

Accessor methods can perform additional functions, such as input validation and security manager checks, before manipulating the state.

Noncompliant Code Example (Public Mutable Field)

Programmers often incorrectly assume that declaring a field or variable final makes the referenced object immutable. Declaring variables that have a primitive type final does prevent changes to their values after initialization (by normal Java processing). However, when the field has a reference type, declaring the field final only makes the reference itself immutable. The final clause has no effect on the referenced object. According to *The Java Language Specification*, §4.12.4, "final Variables" [JLS 2015],

> *If a `final` variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object.*

This noncompliant code example declares a static final mutable hash map with public accessibility:

```
public static final HashMap<Integer, String> hm = new HashMap<Integer, String>();
```

Compliant Solution (Private Mutable Fields)

Mutable fields must be declared private:

```
private static final HashMap<Integer, String> hm = new HashMap<Integer, String>();

public static String getElement(int key) {
  return hm.get(key);
}
```

Depending on the required functionality, accessor methods may return a *copy* of the `HashMap` or a value contained by the `HashMap`. This compliant solution adds an accessor method that returns the value of an element given its key in the `HashMap`. Make sure that you do not return references to private mutable objects from accessor methods (see OBJ05-J. Do not return references to private mutable class members for details).

Noncompliant Code Example (Public Final Array)

A nonzero-length array is always mutable. Declaring a public final array is a potential security risk as the array elements may be modified by a client.

```
public static final String[] items = {/* ... */};
```

Declaring the array reference final prevents modification of the reference but does not prevent clients from modifying the contents of the array.

Compliant Solution (Index Getter)

This compliant solution declares a private array and provides public methods to get individual items and array size:

```
private static final String[] items = {/* ... */};

public static final String getItem(int index) {
  return items[index];
}

public static final int getItemCount() {
  return items.length;
}
```

Providing direct access to the array objects themselves is safe because `String` is immutable.

Compliant Solution (Clone the Array)

This compliant solution defines a private array and a public method that returns a copy of the array:

```
private static final String[] items = {/* ... */};

public static final String[] getItems() {
  return items.clone();
}
```

Because a copy of the array is returned, the original array values cannot be modified by a client. Note that a manual deep copy could be required when dealing with arrays of objects. This generally happens when the objects do not export a `clone()` method (see OBJ06-J. Defensively copy mutable inputs and mutable internal components for more information).

As before, this method provides direct access to the array objects themselves, but this is safe because `String` is immutable. If the array contained mutable objects, the `getItems()` method should return an array of cloned objects instead.

Compliant Solution (Unmodifiable Wrappers)

This compliant solution constructs a public immutable list from the private array. It is safe to share immutable objects without risk that the recipient can modify them [Mettler 2010].

```
private static final String[] items = { ... };

public static final List<String> itemsList =
  Collections.unmodifiableList(Arrays.asList(items));
```

Neither the original array values nor the public list can be modified by a client. For more details about unmodifiable wrappers, refer to OBJ56-J. Provide sensitive mutable classes with unmodifiable wrappers. This solution can also be used when the array contains mutable objects.

Exceptions

OBJ01-J-EX0: Fields with no associated behavior or invariants can be public. According to Sun's Code Conventions document [Conventions 2009]:

> *One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, if you would have used a `struct` instead of a class (if Java supported `struct`), then it's appropriate to make the class's instance variables `public`.*

OBJ01-J-EX1: Fields in a package-private class or in a private nested class may be pubic or protected. There is nothing inherently wrong with declaring fields to be public or protected in these cases. Eliminating accessor methods generally improves the readability of the code both in the class definition and in the client [Bloch 2008].

OBJ01-J-EX2: Static final fields that contain or reference immutable constants may be public or protected.

Risk Assessment

Failing to limit field accessibility can defeat encapsulation, allow attackers to manipulate fields to violate class invariants, or allow these fields to be corrupted as the result of concurrent accesses from multiple threads.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OBJ01-J	Medium	Likely	Medium	P12	L1

Automated Detection

Detection of public and protected fields is trivial; heuristic detection of the presence or absence of accessor methods is straightforward. However, simply reporting all detected cases without suppressing those cases covered by the exceptions to this rule would produce excessive false positives. Sound detection and application of the exceptions to this rule is infeasible; however, heuristic techniques may be useful.

Tool	Version	Checker	Description
SonarQube	6.7	S2386	**Mutable fields should not be "public static"** Implemented for *public static* array, *Collection, Date, and awt.Point* members.

Related Guidelines

SEI CERT C++ Coding Standard	VOID OOP00-CPP. Declare data members private
MITRE CWE	CWE-766, Critical Variable Declared Public
Secure Coding Guidelines for Java SE, Version 5.0	Guideline 6-8 / MUTABLE-8: Define wrapper methods around modifiable internal state

Bibliography

[Bloch 2008]	Item 13, "Minimize the Accessibility of Classes and Members" Item 14, "In Public Classes, Use Accessor Methods, Not Public Fields"
[Conventions 2009]	
[Core Java 2004]	Chapter 6, "Interfaces and Inner Classes"

[JLS 2015]	§4.12.4, "final Variables" §6.6, "Access Control"
[Long 2005]	Section 2.2, "Public Fields"
[Mettler 2010]	Class Properties for Security Review in an Object-Capability Subset of Java