TSM03-J. Do not publish partially initialized objects

During initialization of a shared object, the object must be accessible only to the thread constructing it. However, the object can be published safely (that is, made visible to other threads) once its initialization is complete. The Java memory model (JMM) allows multiple threads to observe the object after its initialization has begun but before it has concluded. Consequently, programs must prevent publication of partially initialized objects.

This rule prohibits publishing a reference to a partially initialized member object instance before initialization has concluded. It specifically applies to safety in multithreaded code. TSM01-J. Do not let the this reference escape during object construction prohibits the `this` reference of the current object from escaping its constructor. OBJ11-J. Be wary of letting constructors throw exceptions describes the consequences of publishing partially initialized objects even in single-threaded programs.

Noncompliant Code Example

This noncompliant code example constructs a `Helper` object in the `initialize()` method of the `Foo` class. The `Helper` object's fields are initialized by its constructor.

```
class Foo {
  private Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

public class Helper {
  private int n;

  public Helper(int n) {
    this.n = n;
  }
  // ...
}
```

If a thread were to access `helper` using the `getHelper()` method before the `initialize()` method executed, the thread would observe an uninitialized `helper` field. Later, if one thread calls `initialize()` and another calls `getHelper()`, the second thread could observe one of the following:

- The `helper` reference as `null`
- A fully initialized `Helper` object with the `n` field set to 42
- A partially initialized `Helper` object with an uninitialized `n`, which contains the default value `0`

In particular, the JMM permits compilers to allocate memory for the new `Helper` object and to assign a reference to that memory to the `helper` field before initializing the new `Helper` object. In other words, the compiler can reorder the write to the `helper` instance field and the write that initializes the `Helper` object (that is, `this.n = n`) so that the former occurs first. This can expose a race window during which other threads can observe a partially initialized `Helper` object instance.

There is a separate issue: if more than one thread were to call `initialize()`, multiple `Helper` objects would be created. This is merely a performance issue—correctness would be preserved. The `n` field of each object would be properly initialized and the unused `Helper` object (or objects) would eventually be garbage-collected.

Compliant Solution (Synchronization)

Appropriate use of method synchronization can prevent publication of references to partially initialized objects, as shown in this compliant solution:

```
class Foo {
  private Helper helper;

  public synchronized Helper getHelper() {
    return helper;
  }

  public synchronized void initialize() {
    helper = new Helper(42);
  }
}
```

Synchronizing both methods guarantees that they cannot execute concurrently. If one thread were to call `initialize()` just before another thread called `getHelper()`, the synchronized `initialize()` method would always finish first. The `synchronized` keywords establish a happens-before relationship between the two threads. Consequently, the thread calling `getHelper()` would see either the fully initialized `Helper` object or an absent `Helper` object (that is, `helper` would contain a null reference). This approach guarantees proper publication both for immutable and mutable members.

Compliant Solution (Final Field)

The JMM guarantees that the fully initialized values of fields that are declared final are safely published to every thread that reads those values at some point no earlier than the end of the object's constructor.

```
class Foo {
  private final Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public Foo() {
    // Point 1
    helper = new Helper(42);
    // Point 2
  }
}
```

However, this solution requires the assignment of a new `Helper` instance to `helper` from Foo's constructor. According to *The Java Language Specification*, §17.5.2, "Reading Final Fields During Construction" [JLS 2015]:

> *A read of a final field of an object within the thread that constructs that object is ordered with respect to the initialization of that field within the constructor by the usual happens-before rules. If the read occurs after the field is set in the constructor, it sees the value the final field is assigned; otherwise, it sees the default value.*

Consequently, the reference to the `helper` instance should remain unpublished until the `Foo` class's constructor has completed (see TSM01-J. Do not let the this reference escape during object construction for additional information).

Compliant Solution (Final Field and Thread-Safe Composition)

Some collection classes provide thread-safe access to contained elements. When a `Helper` object is inserted into such a collection, it is guaranteed to be fully initialized before its reference is made visible. This compliant solution encapsulates the `helper` field in a `Vector<Helper>`.

```
class Foo {
  private final Vector<Helper> helper;

  public Foo() {
    helper = new Vector<Helper>();
  }

  public Helper getHelper() {
    if (helper.isEmpty()) {
      initialize();
    }
    return helper.elementAt(0);
  }

  public synchronized void initialize() {
    if (helper.isEmpty()) {
      helper.add(new Helper(42));
    }
  }
}
```

The `helper` field is declared final to guarantee that the vector is always created before any accesses take place. It can be initialized safely by invoking the synchronized `initialize()` method, which ensures that only one `Helper` object is ever added to the vector. If invoked before `initialize()`, the `getHelper()` avoids the possibility of a null-pointer dereference by conditionally invoking `initialize()`. Although the `isEmpty()` call in `getHelper()` is made from an unsynchronized context (which permits multiple threads to decide that they must invoke `initialize`) race conditions that could result in addition of a second object to the vector are nevertheless impossible. The synchronized `initialize()` method also checks whether `helper` is empty before adding a new `Helper` object, and at most one thread can execute `initialize()` at any time. Consequently, only the first thread to execute `initialize()` can ever see an empty vector and the `getHelper()` method can safely omit any synchronization of its own.

Compliant Solution (Static Initialization)

In this compliant solution, the `helper` field is initialized statically, ensuring that the object referenced by the field is fully initialized before its reference becomes visible:

```
// Immutable Foo
final class Foo {
  private static final Helper helper = new Helper(42);

  public static Helper getHelper() {
    return helper;
  }
}
```

The `helper` field should be declared final to document the class's immutability.

According to JSR-133, Section 9.2.3, "Static Final Fields" [JSR-133 2004]:

> *The rules for class initialization ensure that any thread that reads a `static` field will be synchronized with the static initialization of that class, which is the only place where static final fields can be set. Thus, no special rules in the JMM are needed for static final fields.*

Compliant Solution (Immutable Object - Final Fields, Volatile Reference)

The JMM guarantees that any final fields of an object are fully initialized before a published object becomes visible [Goetz 2006a]. By declaring n final, the `Helper` class is made immutable. Furthermore, if the `helper` field is declared volatile in compliance with VNA01-J. Ensure visibility of shared references to immutable objects, `Helper`'s reference is guaranteed to be made visible to any thread that calls `getHelper()` only after `Helper` has been fully initialized.

```
class Foo {
  private volatile Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

// Immutable Helper
public final class Helper {
  private final int n;

  public Helper(int n) {
    this.n = n;
  }
  // ...
}
```

This compliant solution requires that `helper` be declared volatile and that class `Helper` is immutable. If the `helper` field were not volatile, it would violate VNA01-J. Ensure visibility of shared references to immutable objects.

Providing a public static factory method that returns a new instance of `Helper` is both permitted and encouraged. This approach allows the `Helper` instance to be created in a private constructor.

Compliant Solution (Mutable Thread-Safe Object, Volatile Reference)

When `Helper` is mutable but thread-safe, it can be published safely by declaring the `helper` field in the `Foo` class volatile:

```
class Foo {
  private volatile Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

// Mutable but thread-safe Helper
public class Helper {
  private volatile int n;
  private final Object lock = new Object();

  public Helper(int n) {
    this.n = n;

  }

  public void setN(int value) {
    synchronized (lock) {
      n = value;
    }
  }
}
```

Synchronization is required to ensure the visibility of mutable members after initial publication because the `Helper` object can change state after its construction. This compliant solution synchronizes the `setN()` method to guarantee the visibility of the `n` field.

If the `Helper` class were synchronized incorrectly, declaring `helper` volatile in the `Foo` class would guarantee only the visibility of the initial publication of `Helper`; the visibility guarantee would exclude visibility of subsequent state changes. Consequently, volatile references alone are inadequate for publishing objects that are not thread-safe.

If the `helper` field in the `Foo` class is not declared volatile, the `n` field must be declared volatile to establish a happens-before relationship between the initialization of `n` and the write of `Helper` to the `helper` field. This is required only when the caller (class `Foo`) cannot be trusted to declare `helper` volatile.

Because the `Helper` class is declared public, it uses a private lock to handle synchronization in conformance with LCK00-J. Use private final lock objects to synchronize classes that may interact with untrusted code.

Exceptions

TSM03-J-EX0: Classes that prevent partially initialized objects from being used may publish partially initialized objects. This could be implemented, for example, by setting a volatile Boolean flag in the last statement of the initializing code and checking whether the flag is set before allowing class methods to execute.

The following compliant solution shows this technique:

```
public class Helper {
  private int n;
  private volatile boolean initialized; // Defaults to false

  public Helper(int n) {
    this.n = n;
    this.initialized = true;
  }

  public void doSomething() {
    if (!initialized) {
      throw new SecurityException(
          "Cannot use partially initialized instance");
    }
    // ...
  }
  // ...
}
```

This technique ensures that if a reference to the `Helper` object instance were published before its initialization was complete, the instance would be unusable because each method within `Helper` checks the flag to determine whether the initialization has finished.

Risk Assessment

Failure to synchronize access to shared mutable data can cause different threads to observe different states of the object or to observe a partially initialized object.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
TSM03-J	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.1p0	**FB.MT_CORRECTNESS.DC_PARTIALLY_CONSTRUCTED**	Possible exposure of partially initialized object

Bibliography

[API 2006]	
[Bloch 2001]	Item 48, "Synchronize Access to Shared Mutable Data"
[Goetz 2006a]	Section 3.5.3, "Safe Publication Idioms"
[Goetz 2007]	Pattern #2, "One-Time Safe Publication"
[JPL 2006]	Section 14.10.2, "Final Fields and Security"
[Pugh 2004]