OBJ11-J. Be wary of letting constructors throw exceptions

An object is partially initialized if a constructor has begun building the object but has not finished. As long as the object is not fully initialized, it must be hidden from other classes.

Other classes might access a partially initialized object from concurrently running threads. This rule is a specific instance of TSM01-J. Do not let the this reference escape during object construction but focuses only on single-threaded programs. Multithreaded programs must also comply with TSM03-J. Do not publish partially initialized objects.

Some uses of variables require *failure atomicity*. This requirement typically arises when a variable constitutes an aggregation of different objects, for example, a composition-and-forwarding-based approach, as described in OBJ02-J. Preserve dependencies in subclasses when changing superclasses. In the absence of failure atomicity, the object can be left in an inconsistent state as a result of partial initialization.

There are three common approaches to dealing with the problem of partially initialized objects:

- **Exception in constructor.** This approach throws an exception in the object's constructor. Unfortunately, an attacker can maliciously obtain the instance of such an object. For example, an attack that uses the finalizer construct allows an attacker to invoke arbitrary methods within the class, even if the class methods are protected by a security manager.
- **Final field.** Declaring the variable that is initialized to the object as `final` prevents the object from being partially initialized. The compiler produces a warning when there is a possibility that the variable's object might not be fully initialized. Declaring the variable `fina` also guarantees initialization safety in multithreaded code. According to *The Java Language Specification* (JLS), §17.5, "`final` Field Semantics" [JLS 2015], "An object is considered to be completely initialized when its constructor finishes. A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's final fields." In other words, when a constructor executing in one thread initializes a final field to a known safe value, other threads are unable to see the *preinitialized* values of the object.
- **Initialized flag.** This approach allows uninitialized or partially initialized objects to exist in a known failed state; such objects are commonly known as *zombie objects*. This solution is error prone because any access to such a class must first check whether the object has been correctly initialized.

The following table summarizes these three approaches:

Solution	Uninitialized Values	Partially Initialized Objects
Exception in constructor	Prevents	Does not prevent
Final field	Prevents	Prevents
Initialized flag	Detects	Detects

Noncompliant Code Example (Finalizer Attack)

This noncompliant code example, based on an example by Kabutz [Kabutz 2001], defines the constructor of the `BankOperations` class so that it performs social security number (SSN) verification using the method `performSSNVerification()`. The implementation of the `performSSNVerification()` method assumes that an attacker does not know the correct SSN and trivially returns false.

```
public class BankOperations {
  public BankOperations() {
    if (!performSSNVerification()) {
      throw new SecurityException("Access Denied!");
    }
  }

  private boolean performSSNVerification() {
    return false; // Returns true if data entered is valid, else false
                  // Assume that the attacker always enters an invalid SSN
  }

  public void greet() {
    System.out.println("Welcome user! You may now use all the features.");
  }
}

public class Storage {
  private static BankOperations bop;

  public static void store(BankOperations bo) {
  // Store only if it is initialized
    if (bop == null) {
      if (bo == null) {
        System.out.println("Invalid object!");
        System.exit(1);
      }
      bop = bo;
    }
  }
}

public class UserApp {
  public static void main(String[] args) {
    BankOperations bo;
    try {
      bo = new BankOperations();
    } catch (SecurityException ex) { bo = null; }

    Storage.store(bo);
    System.out.println("Proceed with normal logic");
  }
}
```

The constructor throws a `SecurityException` when SSN verification fails. The `UserApp` class appropriately catches this exception and displays an "Access Denied" message. However, these precautions fail to prevent a malicious program from invoking methods of the partially initialized class `BankOperations`, as shown by the following exploit code.

The goal of the attack is to capture a reference to the partially initialized object of the `BankOperations` class. If a malicious subclass catches the `SecurityException` thrown by the `BankOperations` constructor, it is unable to further exploit the vulnerable code because the new object instance has gone out of scope. Instead, an attacker can exploit this code by extending the `BankOperations` class and overriding the `finalize()` method. This attack intentionally violates MET12-J. Do not use finalizers.

When the constructor throws an exception, the garbage collector waits to grab the object reference. However, the object cannot be garbage-collected until *after* the finalizer completes its execution. The attacker's finalizer obtains and stores a reference by using the `this` keyword. Consequently, the attacker can maliciously invoke any instance method on the base class by using the stolen instance reference. This attack can even bypass a check by a security manager.

```
public class Interceptor extends BankOperations {
  private static Interceptor stealInstance = null;

  public static Interceptor get() {
    try {
      new Interceptor();
    } catch (Exception ex) {/* Ignore exception */}
    try {
      synchronized (Interceptor.class) {
        while (stealInstance == null) {
          System.gc();
          Interceptor.class.wait(10);
        }
      }
    } catch (InterruptedException ex) { return null; }
    return stealInstance;
  }

  public void finalize() {
    synchronized (Interceptor.class) {
      stealInstance = this;
      Interceptor.class.notify();
    }
    System.out.println("Stole the instance in finalize of " + this);
  }
}

public class AttackerApp { // Invoke class and gain access
                           // to the restrictive features
  public static void main(String[] args) {
    Interceptor i = Interceptor.get(); // Stolen instance

    // Can store the stolen object even though this should have printed
    // "Invalid Object!"
    Storage.store(i);

    // Now invoke any instance method of BankOperations class
    i.greet();

    UserApp.main(args); // Invoke the original UserApp
  }
}
```

Compliance with ERR00-J. Do not suppress or ignore checked exceptions and ERR03-J. Restore prior object state on method failure can help to ensure that fields are appropriately initialized in catch blocks. A developer who explicitly initializes the variable to `null` is more likely to document this behavior so that other programmers or clients include the appropriate null reference checks where required. Moreover, this approach guarantees initialization safety in a multithreaded scenario.

Compliant Solution (Final)

This compliant solution declares the partially initialized class final so that it cannot be extended:

```
public final class BankOperations {
  // ...
}
```

Compliant Solution (Final `finalize()`)

If the class itself cannot be declared final, it can still thwart the finalizer attack by declaring its own `finalize()` method and making it final:

```
public class BankOperations {
  public final void finalize() {
    // Do nothing
  }
}
```

This solution is allowed under exception MET12-J-EX1, which permits a class to use an empty final finalizer to prevent a finalizer attack.

Compliant Solution (Java SE 6, Public and Private Constructors)

This compliant solution applies to Java SE 6 and later versions in which the JVM will not execute an object's finalizer if the object's constructor throws an exception before the `java.lang.Object` constructor exits [SCG 2009]. Developers can use this feature to throw an exception in a constructor without risking the escape of a partially initialized object (via the finalizer attack described previously). However, doing so requires a careful coding of the constructor because Java ensures that the `java.lang.Object` constructor executes on or before the first statement of any constructor. If the first statement in a constructor is a call to either a superclass's constructor or another constructor in the same class, then the `java.lang.Object` constructor will be executed somewhere in that call. Otherwise, Java will execute the default constructor of the superclass before any of the constructor's code and the `java.lang.Object` constructor will be executed through that (implicit) invocation.

Consequently, to execute potentially exception-raising checks *before* the `java.lang.Object` constructor exits, the programmer must place them in an argument expression of an explicit constructor invocation. For example, the single constructor can be split into three parts: a public constructor whose interface remains unchanged, a private constructor that takes (at least) one argument and performs the actual work of the original constructor, and a method that performs the checks. The public constructor invokes the private constructor on its first line while invoking the method as an argument expression. All code in the expression will be executed before the private constructor, ensuring that any exceptions will be raised before the `java.lang.Object` constructor is invoked.

This compliant solution demonstrates the design. Note that the `performSSNVerification()` method is modified to throw an exception rather than returning false if the security check fails.

```
public class BankOperations {
  public BankOperations() {
    this(performSSNVerification());
  }

  private BankOperations(boolean secure) {
    // secure is always true
    // Constructor without any security checks
  }

  private static boolean performSSNVerification() {
    // Returns true if data entered is valid, else throws a
SecurityException
    // Assume that the attacker just enters invalid SSN, so this method
always throws the exception
    throw new SecurityException("Invalid SSN!");
  }

  // ...remainder of BankOperations class definition
}
```

Compliant Solution (Initialized Flag)

Rather than throwing an exception, this compliant solution uses an *initialized flag* to indicate whether an object was successfully constructed. The flag is initialized to false and set to true when the constructor finishes successfully.

```
class BankOperations {
  private volatile boolean initialized = false;

  public BankOperations() {
    if (!performSSNVerification()) {
      return;               // object construction failed
    }

    this.initialized = true; // Object construction successful
  }

  private boolean performSSNVerification() {
    return false;
  }

  public void greet() {
    if (!this.initialized) {
      throw new SecurityException("Invalid SSN!");
    }

    System.out.println(
        "Welcome user! You may now use all the features.");
  }
}
```

The initialized flag prevents any attempt to access the object's methods if the object is not fully constructed. Because each method must check the initialized flag to detect a partially constructed object, this solution imposes a speed penalty on the program. It is also harder to maintain because it is easy for a maintainer to add a method that fails to check the initialized flag.

According to Charlie Lai [Lai 2008]:

> *If an object is only partially initialized, its internal fields likely contain safe default values such as `null`. Even in an untrusted environment, such an object is unlikely to be useful to an attacker. If the developer deems the partially initialized object state secure, then the developer doesn't have to pollute the class with the flag. The flag is necessary only when such a state isn't secure or when accessible methods in the class perform sensitive operations without referencing any internal field.*

The initialized flag is volatile to ensure that the setting of the flag to true happens-before any reads of the variable. The current code does not allow for multiple threads to read the field before the constructor terminates, but this object could always be subclassed and run in an environment where multiple threads can access the variable.

Noncompliant Code Example (Static Variable)

This noncompliant code example uses a nonfinal static variable. The JLS does not mandate complete initialization and safe publication even though a static initializer has been used. Note that in the event of an exception during initialization, the variable can be incorrectly initialized.

```
class Trade {
  private static Stock s;
  static {
    try {
      s = new Stock();
    } catch (IOException e) {
      /* Does not initialize s to a safe state */
    }
  }
  // ...
}
```

Compliant Solution (Final Static Variable)

This compliant solution guarantees safe publication by declaring the `Stock` field final:

```
private static final Stock s;
```

Unlike the previous compliant solution, however, this approach permits a possibly null value but guarantees that a non-null value refers to a completely initialized object.

Risk Assessment

Allowing access to a partially initialized object can provide an attacker with an opportunity to resurrect the object before or during its finalization; as a result, the attacker can bypass security checks.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
OBJ11-J	High	Probable	Medium	P12	L1

Automated Detection

Automated detection for this rule is infeasible in the general case. Some instances of nonfinal classes whose constructors can throw exceptions could be straightforward to diagnose.

Tool	Version	Checker	Description
Parasoft Jtest	10.3	**EXCEPT.ENFC**	Implemented

Related Vulnerabilities

CVE-2008-5353 describes a collection of vulnerabilities in Java. In one of the vulnerabilities, an applet causes an object to be deserialized using `ObjectInputStream.readObject()`, but the input is controlled by an attacker. The object actually read is a serializable subclass of `ClassLoader`, and it has a `readObject()` method that stashes the object instance into a static variable; consequently, the object survives the serialization. As a result, the applet manages to construct a `ClassLoader` object by passing the restrictions against this in an applet, and the `ClassLoader` allows it to construct classes that are not subject to the security restrictions of an applet. This vulnerability is described in depth in SER08-J. Minimize privileges before deserializing from a privileged context.

Related Guidelines

Secure Coding Guidelines for Java SE, Version 5.0	Guideline 4-5 / EXTEND-5: Limit the extensibility of classes and methods Guideline 7-3 / OBJECT-3: Defend against partially initialized instances of non-final classes

Bibliography

[API 2006]	`finalize()`
[Darwin 2004]	Section 9.5, "The Finalize Method"
[Flanagan 2005]	Section 3.3, "Destroying and Finalizing Objects"
[JLS 2015]	§8.3.1, Field Modifiers §12.6, Finalization of Class Instances §17.5, "`final` Field Semantics"
[Kabutz 2001]	Issue 032, "Exceptional Constructors—Resurrecting the Dead"
[Lai 2008]	"Java Insecurity: Accounting for Subtleties That Can Compromise Code"
[Masson 2011]	"Secure Your Code against the Finalizer Vulnerability"