Chapter 2

Defensive Programming

Defensive programming is carefully guarded programming that helps you to construct reliable software by designing each component to protect itself as much as possible: for example, by checking that undocumented assumptions remain valid [Goodliffe 2007]. The guidelines in this chapter address areas of the Java language that can help to constrain the effect of an error or help to recover from an error.

Java language mechanisms should be used to limit the scope, lifetime, and accessibility of program resources. Also, Java annotations can be used to document the program, aiding readability and maintenance. Java programmers should be aware of implicit behaviors and avoid unwarranted assumptions about how the system behaves.

A good overall principle for defensive programming is simplicity. A complicated system is difficult to understand, difficult to maintain, and difficult to get right in the first place. If a construct turns out to be complicated to implement, consider redesigning or refactoring it to reduce the complexity.

Finally, the program should be designed to be as robust as possible. Wherever possible, the program should help the Java runtime system by limiting the resources it uses and by releasing acquired resources when they are no longer needed. Again, this can often be achieved by limiting the lifetime and accessibility of objects and other programming constructs. Not all eventualities can be anticipated, so a strategy should be developed to provide a graceful exit of last resort.

■ 22. Minimize the scope of variables

Scope minimization helps developers avoid common programming errors, improves code readability by connecting the declaration and actual use of a variable, and improves maintainability because unused variables are more easily detected and removed. It may also allow objects to be recovered by the garbage collector more quickly, and it prevents violations of Guideline 37, "Do not shadow or obscure identifiers in subscopes."

Noncompliant Code Example

This noncompliant code example shows a variable that is declared outside the `for` loop.

```
public class Scope {
  public static void main(String[] args) {
    int i = 0;
    for (i = 0; i < 10; i++) {
      // Do operations
    }
  }
}
```

This code is noncompliant because, even though variable `i` is not intentionally used outside the `for` loop, it is declared in method scope. One of the few scenarios where variable `i` needs to be declared in method scope is when the loop contains a break statement, and the value of `i` must be inspected after conclusion of the loop.

Compliant Solution

Minimize the scope of variables where possible. For example, declare loop indices within the `for` statement:

```
public class Scope {
  public static void main(String[] args) {
    for (int i = 0; i < 10; i++) { // Contains declaration
      // Do operations
    }
  }
}
```

Noncompliant Code Example

This noncompliant code example shows a variable `count` that is declared outside the `counter()` method, although the variable is not used outside the `counter()` method.

```
public class Foo {
  private int count;
  private static final int MAX_COUNT = 10;

  public void counter() {
    count = 0;
    while (condition()) {
      /* ... */
      if (count++ > MAX_COUNT) {
        return;
      }
    }
  }

  private boolean condition() {/* ... */}
  // No other method references count
  // but several other methods reference MAX_COUNT
}
```

The reusability of the method is reduced because if the method were copied to another class, then the `count` variable would also need to be redefined in the new context. Furthermore, the analyzability of the `counter` method would be reduced, as whole program data flow analysis would be necessary to determine possible values for `count.`

Compliant Solution

In this compliant solution, the `count` field is declared local to the `counter()` method:

```
public class Foo {
  private static final int MAX_COUNT = 10;

  public void counter() {
    int count = 0;
    while (condition()) {
      /* ... */
      if (count++ > MAX_COUNT) {
        return;
      }
    }
  }
```

```
  private boolean condition() {/* ... */}
  // No other method references count
  // but several other methods reference MAX_COUNT
}
```

Applicability

Detecting local variables that are declared in a larger scope than is required by the code as written is straightforward and can eliminate the possibility of false positives.

Detecting multiple `for` statements that use the same index variable is straightforward; it produces false positives only in the unusual case where the value of the index variable is intended to persist between loops.

Bibliography

[Bloch 2001] Item 29, "Minimize the Scope of Local Variables"

[JLS 2013] §14.4, "Local Variable Declaration Statements"

■ 23. Minimize the scope of the `@SuppressWarnings` annotation

When the compiler detects potential type-safety issues arising from mixing raw types with generic code, it issues *unchecked warnings*, including *unchecked cast warnings, unchecked method invocation warnings, unchecked generic array creation warnings*, and *unchecked conversion warnings* [Bloch 2008]. It is permissible to use the `@SuppressWarnings("unchecked")` annotation to suppress unchecked warnings when, and only when, the warning-emitting code is guaranteed to be type safe. A common use case is mixing legacy code with new client code. The perils of ignoring unchecked warnings are discussed extensively in *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012], "OBJ03-J. Do not mix generic with nongeneric raw types in new code."

According to the Java API Annotation Type `SuppressWarnings` documentation [API 2013],

> As a matter of style, programmers should always use this annotation on the most deeply nested element where it is effective. If you want to suppress a warning in a particular method, you should annotate that method rather than its class.

The `@SuppressWarnings` annotation can be used in the declaration of variables and methods as well as an entire class. It is, however, important to narrow its scope so that only those warnings that occur in the narrower scope are suppressed.

Noncompliant Code Example

In this noncompliant code example, the `@SuppressWarnings` annotation's scope encompasses the whole class:

```
@SuppressWarnings("unchecked")
class Legacy {
  Set s = new HashSet();
  public final void doLogic(int a, char c) {
    s.add(a);
    s.add(c); // Type-unsafe operation, ignored
  }
}
```

This code is dangerous because all unchecked warnings within the class are suppressed. Oversights of this nature can result in a `ClassCastException` at runtime.

Compliant Solution

Limit the scope of the `@SuppressWarnings` annotation to the nearest code that generates a warning. In this case, it may be used in the declaration for the `Set`:

```
class Legacy {
  @SuppressWarnings("unchecked")
  Set s = new HashSet();
  public final void doLogic(int a, char c) {
    s.add(a); // Produces unchecked warning
    s.add(c); // Produces unchecked warning
  }
}
```

Noncompliant Code Example (`ArrayList`)

This noncompliant code example is from an old implementation of `java.util.ArrayList`:

```
@SuppressWarnings("unchecked")
public <T> T[] toArray(T[] a) {
  if (a.length < size) {
    // Produces unchecked warning
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
  }
  // ...
}
```

When the class is compiled, it emits an unchecked cast warning:

```
// Unchecked cast warning
ArrayList.java:305: warning: [unchecked] unchecked cast found :
  Object[], required: T[]
return (T[]) Arrays.copyOf(elements, size, a.getClass());
```

This warning cannot be suppressed for just the `return` statement because it is not a declaration [JLS 2013]. As a result, the programmer suppresses warnings for the entire method. This can cause issues when functionality that performs type-unsafe operations is added to the method at a later date [Bloch 2008].

Compliant Solution (`ArrayList`)

When it is impossible to use the `@SuppressWarnings` annotation in an appropriate scope, as in the preceding noncompliant code example, declare a new variable to hold the return value and adorn it with the `@SuppressWarnings` annotation.

```
// ...
@SuppressWarnings("unchecked")
T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
return result;
// ...
```

Applicability

Failure to reduce the scope of the `@SuppressWarnings` annotation can lead to run-time exceptions and break type-safety guarantees.

This rule cannot be statically enforced in full generality; however, static analysis can be used for some special cases.

Bibliography

[API 2013]	Annotation Type `SuppressWarnings`
[Bloch 2008]	Item 24, "Eliminate Unchecked Warnings"
[Long 2012]	OBJ03-J. Do not mix generic with nongeneric raw types in new code

■ 24. Minimize the accessibility of classes and their members

Classes and class members (classes, interfaces, fields, and methods) are access-controlled in Java. Access is indicated by an access modifier (`public`, `protected`, or `private`) or by the absence of an access modifier (the default access, also called *package-private access*).

Table 2–1. Access control rules

Access Specifier	Class	Package	Subclass	World
`private`	x			
None	x	x	x*	
`protected`	x	x	x**	
`public`	x	x	x	x

*Subclasses within the same package can also access members that lack access specifiers (default or package-private visibility). An additional requirement for access is that the subclasses must be loaded by the class loader that loaded the class containing the package-private members. Subclasses in a different package cannot access such package-private members.

**To reference a protected member, the accessing code must be contained in either the class that defines the protected member or in a subclass of that defining class. Subclass access is permitted without regard to the package location of the subclass.

Table 2–1 presents a simplified view of the access control rules. An *x* indicates that the particular access is permitted from within that domain. For example, an *x* in the class column means that the class member is accessible to code present within the same class in which it is declared. Similarly, the package column indicates that the member is accessible from any class (or subclass) defined in the same package, provided that the class (or subclass) is loaded by the class loader that loaded the class containing the member. The same class loader condition applies only to package-private member access.

Classes and class members must be given the minimum possible access so that malicious code has the least opportunity to compromise security. As far as possible, classes should avoid exposing methods that contain (or invoke) sensitive code through interfaces; interfaces allow only publicly accessible methods, and such methods are part of the public application programming interface (API) of the class. (Note that this is the opposite of Joshua Bloch's recommendation to prefer interfaces for APIs [Bloch 2008, Item 16].) One exception to this is implementing an *unmodifiable* interface that exposes a public immutable view of a mutable object. (See *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012], "OBJ04-J. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code.") Note that even if a nonfinal class's visibility is default, it can be susceptible to misuse if it contains public methods. Methods that perform all necessary security checks and sanitize all inputs may be exposed through interfaces.

Protected accessibility is invalid for non-nested classes, but nested classes may be declared protected. Fields of nonfinal public classes should rarely be declared

protected; untrusted code in another package can subclass the class, and access the member. Furthermore, protected members are part of the API of the class, and consequently require continued support. When this rule is followed, declaring fields as protected is unnecessary. "OBJ01-J. Declare data members as private and provide accessible wrapper methods" [Long 2012] recommends declaring fields as private.

If a class, interface, method, or field is part of a published API, such as a web service endpoint, it may be declared public. Other classes and members should be declared either package-private or private. For example, non-security-critical classes are encouraged to provide public static factories to implement instance control with a private constructor.

Noncompliant Code Example (Public Class)

This noncompliant code example defines a class that is internal to a system and not part of any public API. Nonetheless, this class is declared public.

```
public final class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Even though this example complies with "OBJ01-J. Declare data members as private and provide accessible wrapper methods" [Long 2012], untrusted code could instantiate `Point` and invoke the public `getPoint()` method to obtain the coordinates.

Compliant Solution (Final Classes with Public Methods)

This compliant solution declares the `Point` class as package-private in accordance with its status as not part of any public API:

```
final class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

A top-level class, such as `Point`, cannot be declared private. Package-private accessibility is acceptable, provided package insertion attacks are avoided. (See "ENV01-J. Place all security-sensitive code in a single JAR and sign and seal it" [Long 2012].) A package insertion attack occurs when, at runtime, any protected or package-private members of a class can be called directly by a class that is maliciously inserted into the same package. However, this attack is difficult to carry out in practice because, in addition to the requirement of infiltrating the package, the target and the untrusted class must be loaded by the same class loader. Untrusted code is typically deprived of such levels of access.

Because the class is final, the `getPoint()` method can be declared public. A public subclass that violates this rule cannot override the method and expose it to untrusted code, so its accessibility is irrelevant. For nonfinal classes, reducing the accessibility of methods to private or package-private eliminates this threat.

Compliant Solution (Nonfinal Classes with Nonpublic Methods)

This compliant solution declares the `Point` class and its `getPoint()` method as package-private, which allows the `Point` class to be nonfinal and allows `getPoint()` to be invoked by classes present within the same package and loaded by a common class loader:

```
class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
```

```
  void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Noncompliant Code Example (Public Class with Public Static Method)

This noncompliant code example again defines a class that is internal to a system and not part of any public API. Nonetheless, this class is declared public.

```
public final class Point {
  private static final int x = 1;
  private static final int y = 2;

  private Point(int x, int y) {}

  public static void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

This example also complies with "OBJ01-J. Declare data members as private and provide accessible wrapper methods" [Long 2012], untrusted code could access `Point` and invoke the public static `getPoint()` to obtain the default coordinates. The attempt to implement instance control using a private constructor is futile because the public static method exposes internal class contents.

Compliant Solution (Package-Private Class)

This compliant solution reduces the accessibility of the class to package-private.

```
final class Point {
  private static final int x = 1;
  private static final int y = 2;

  private Point(int x, int y) {}

  public static void getPoint() {
    System.out.println("(" + x + "," + y + ")");
  }
}
```

Access to the `getPoint()` method is restricted to classes located within the same package. Untrusted code is prevented from invoking `getPoint()` and obtaining the coordinates.

Applicability

Granting excessive access breaks encapsulation and weakens the security of Java applications.

A system with an API designed for use (and possibly extended) by third-party code must expose the API through a public interface. The demands of such an API override this guideline.

For any given piece of code, the minimum accessibility for each class and member can be computed so as to avoid introducing compilation errors. A limitation is that the result of this computation may lack any resemblance to what the programmer intended when the code was written. For example, unused members can obviously be declared private. However, such members could be unused only because the particular body of code examined coincidentally lacks references to the members. Nevertheless, this computation can provide a useful starting point for a programmer who wishes to minimize the accessibility of classes and their members.

Bibliography

[Bloch 2008]	Item 13, "Minimize the Accessibility of Classes and Members"
	Item 16, "Prefer Interfaces to Abstract Classes"
[Campione 1996]	Access Control
[JLS 2013]	§6.6, "Access Control"
[Long 2012]	ENV01-J. Place all security-sensitive code in a single JAR and sign and seal it
	OBJ01-J. Declare data members as private and provide accessible wrapper methods
	OBJ04-J. Provide mutable classes with copy functionality to safely allow passing instances to untrusted code
[McGraw 1999]	Chapter 3, "Java Language Security Constructs"

25. Document thread-safety and use annotations where applicable

The Java language annotation facility is useful for documenting design intent. Source code annotation is a mechanism for associating metadata with a program element and making it available to the compiler, analyzers, debuggers, or Java Virtual

Machine (JVM) for examination. Several annotations are available for documenting thread-safety or the lack thereof.

Obtaining Concurrency Annotations

Two sets of concurrency annotations are freely available and licensed for use in any code. The first set consists of four annotations described in *Java Concurrency in Practice* (JCIP) [Goetz 2006], which can be downloaded from http://jcip.net. The JCIP annotations are released under the Creative Commons Attribution License.

The second, larger set of concurrency annotations is available from and supported by SureLogic. These annotations are released under the Apache Software License, Version 2.0, and can be downloaded at www.surelogic.com. The annotations can be verified by the SureLogic JSure tool, and they remain useful for documenting code even when the tool is unavailable. These annotations include the JCIP annotations because they are supported by the JSure tool. (JSure also supports use of the JCIP JAR file.)

To use the annotations, download and add one or both of the aforementioned JAR files to the code's build path. The use of these annotations to document thread-safety is described in the following sections.

Documenting Intended Thread-Safety

JCIP provides three class-level annotations to describe the programmer's design intent with respect to thread-safety.

The `@ThreadSafe` annotation is applied to a class to indicate that it is *thread-safe*. This means that no sequences of accesses (reads and writes to public fields, calls to public methods) can leave the object in an inconsistent state, regardless of the interleaving of these accesses by the runtime or any external synchronization or coordination on the part of the caller.

For example, the following `Aircraft` class specifies that it is thread-safe as part of its locking policy documentation. This class protects the `x` and `y` fields using a reentrant lock.

```
@ThreadSafe
@Region("private AircraftState")
@RegionLock("StateLock is stateLock protects AircraftState")
public final class Aircraft {
  private final Lock stateLock = new ReentrantLock();
  // ...
  @InRegion("AircraftState")
  private long x, y;
  // ...
```

```
  public void setPosition(long x, long y) {
    stateLock.lock();
    try {
      this.x = x;
      this.y = y;
    } finally {
      stateLock.unlock();
    }
  }
  // ...
}
```

The `@Region` and `@RegionLock` annotations document the locking policy upon which the promise of thread-safety is predicated.

Even when one or more `@RegionLock` or `@GuardedBy` annotations have been used to document the locking policy of a class, the `@ThreadSafe` annotation provides an intuitive way for reviewers to learn that the class is thread-safe.

The `@Immutable` annotation is applied to *immutable* classes. Immutable objects are inherently thread-safe; once they are fully constructed, they may be published via a reference and shared safely among multiple threads.

The following example shows an immutable `Point` class:

```
@Immutable
public final class Point {
  private final int f_x;
  private final int f_y;

  public Point(int x, int y) {
    f_x = x;
    f_y = y;
  }

  public int getX() {
    return f_x;
  }

  public int getY() {
    return f_y;
  }
}
```

According to Joshua Bloch [Bloch 2008],

> It is not necessary to document the immutability of `enum` types. Unless it is obvious from the return type, static factories must document the thread safety of the returned object, as demonstrated by `Collections.synchronizedMap`.

The `@NotThreadSafe` annotation is applied to classes that are not thread-safe. Many classes fail to document whether they are safe for multithreaded use. Consequently, a programmer has no easy way to determine whether the class is thread-safe. This annotation provides clear indication of the class's lack of thread-safety.

For example, most of the collection implementations provided in `java.util` are not thread-safe. The class `java.util.ArrayList` could document this as follows:

```
package java.util.ArrayList;

@NotThreadSafe
public class ArrayList<E> extends ... {
  // ...
}
```

Documenting Locking Policies

It is important to document all the locks that are being used to protect shared state. According to Brian Goetz and colleagues [Goetz 2006],

> For each mutable state variable that may be accessed by more than one thread, *all* accesses to that variable must be performed with the *same* lock held. In this case, we say that the variable is *guarded by* that lock. (p. 28)

JCIP provides the `@GuardedBy` annotation for this purpose, and SureLogic provides the `@RegionLock` annotation. The field or method to which the `@GuardedBy` annotation is applied can be accessed only when holding a particular lock. It may be an intrinsic lock or a dynamic lock such as `java.util.concurrent.Lock`.

For example, the following `MovablePoint` class implements a movable point that can remember its past locations using the `memo` array list:

```
@ThreadSafe
public final class MovablePoint {

  @GuardedBy("this")
  double xPos = 1.0;
```

```
  @GuardedBy("this")
  double yPos = 1.0;
  @GuardedBy("itself")
  static final List<MovablePoint> memo
    = new ArrayList<MovablePoint>();

  public void move(double slope, double distance) {
    synchronized (this) {
      rememberPoint(this);
      xPos += (1 / slope) * distance;
      yPos += slope * distance;
    }
  }

  public static void rememberPoint(MovablePoint value) {
    synchronized (memo) {
      memo.add(value);
    }
  }
}
```

The `@GuardedBy` annotations on the `xPos` and `yPos` fields indicate that access to these fields is protected by holding a lock on `this`. The `move()` method also synchronizes on `this`, which modifies these fields. The `@GuardedBy` annotation on the `memo` list indicates that a lock on the `ArrayList` object protects its contents. The `rememberPoint()` method also synchronizes on the memo list.

One issue with the `@GuardedBy` annotation is that it fails to indicate when there is a relationship between the fields of a class. This limitation can be overcome by using the SureLogic `@RegionLock` annotation, which declares a new region lock for the class to which this annotation is applied. This declaration creates a new named lock that associates a particular lock object with a region of the class. The region may be accessed only when the lock is held. For example, the `SimpleLock` locking policy indicates that synchronizing on the instance protects all of its state:

```
@RegionLock("SimpleLock is this protects Instance")
class Simple { ... }
```

Unlike `@GuardedBy`, the `@RegionLock` annotation allows the programmer to give an explicit, and hopefully meaningful, name to the locking policy.

In addition to naming the locking policy, the `@Region` annotation allows a name to be given to the region of the state that is being protected. That name makes it clear that the state and locking policy belong together, as demonstrated in the following example:

```
@Region("private AircraftPosition")
@RegionLock("StateLock is stateLock protects AircraftPosition")
public final class Aircraft {
  private final Lock stateLock = new ReentrantLock();

  @InRegion("AircraftPosition")
  private long x, y;

  @InRegion("AircraftPosition")
  private long altitude;
  // ...
  public void setPosition(long x, long y) {
    stateLock.lock();
    try {
      this.x = x;
      this.y = y;
    } finally {
      stateLock.unlock();
    }
  }
  // ...
}
```

In this example, a locking policy named `StateLock` is used to indicate that locking on `stateLock` protects the named `AircraftPosition` region, which includes the mutable state used to represent the position of the aircraft.

Construction of Mutable Objects

Typically, object construction is considered an exception to the locking policy because objects are thread-confined when they are first created. An object is confined to the thread that uses the `new` operator to create its instance. After creation, the object can be published to other threads safely. However, the object is not shared until the thread that created the instance allows it to be shared. Safe publication approaches discussed in *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012], "TSM01-J. Do not let the `this` reference escape during object construction," can be expressed succinctly with the `@Unique("return")` annotation.

For example, in the following code, the `@Unique("return")` annotation documents that the object returned from the constructor is a unique reference:

```
@RegionLock("Lock is this protects Instance")
public final class Example {
  private int x = 1;
  private int y;

  @Unique("return")
  public Example(int y) {
    this.y = y;
  }
  // ...
}
```

Documenting Thread-Confinement Policies

Dean Sutherland and William Scherlis propose annotations that can document thread-confinement policies. Their approach allows verification of the annotations against as-written code [Sutherland 2010].

For example, the following annotations express the design intent that a program has, at most, one Abstract Window Toolkit (AWT) event dispatch thread and several compute threads, and that the compute threads are forbidden to handle AWT data structures or events:

```
@ThreadRole AWT, Compute
@IncompatibleThreadRoles AWT, Compute
@MaxRoleCount AWT 1
```

Documenting Wait–Notify Protocols

According to Goetz and colleagues [Goetz 2006],

> A state-dependent class should either fully expose (and document) its waiting and notification protocols to subclasses, or prevent subclasses from participating in them at all. (This is an extension of "design and document for inheritance, or else prohibit it" [EJ Item 15].) At the very least, designing a state-dependent class for inheritance requires exposing the condition queues and locks and documenting the condition predicates and synchronization policy; it may also require exposing the underlying state variables. (The worst thing a state-dependent class can do is expose its state to subclasses but not document its protocols for waiting and notification; this is like a class exposing its state variables but not documenting its invariants.) (p. 395)

Wait–notify protocols should be documented adequately. Currently, we are not aware of any annotations for this purpose.

Applicability

Annotating concurrent code helps document the design intent and can be used to automate the detection and prevention of race conditions and data races.

Bibliography

[Bloch 2008] Item 70, "Document Thread Safety"

[Goetz 2006] *Java Concurrency in Practice*

[Long 2012] TSM01-J. Do not let the `this` reference escape during object construction

[Sutherland 2010] "Composable Thread Coloring"

■ 26. Always provide feedback about the resulting value of a method

Methods should be designed to return a value that allows the developer to learn about the current state of the object and/or the result of an operation. This advice is consistent with *The CERT® Oracle® Secure Coding Standard for Java™* [Long 2012], "EXP00-J. Do not ignore values returned by methods." The returned value should be representative of the last known state and should be chosen keeping in mind the perceptions and mental model of the developer.

Feedback can also be provided by throwing either standard or custom exception objects derived from the `Exception` class. With this approach, the developer can still get precise information about the outcome of the method and proceed to take the necessary actions. To do so, the exception should provide a detailed account of the abnormal condition at the appropriate abstraction level.

APIs should use a combination of these approaches, both to help clients distinguish correct results from incorrect ones and to encourage careful handling of any incorrect results. In cases where there is a commonly accepted error value that cannot be misinterpreted as a valid return value for the method, that error value should be returned; and in other cases an exception should be thrown. A method must not return a value that can hold both valid return data and an error code; see Guideline 52, "Avoid in-band error indicators," for more details.

Alternatively, an object can provide a state-testing method [Bloch 2008] that checks whether the object is in a consistent state. This approach is useful only in cases where the object's state cannot be modified by external threads. This prevents a

time-of-check, time-of-use (TOCTOU) race condition between invocation of the object's state-testing method and the call to a method that depends on the object's state. During this interval, the object's state could change unexpectedly or even maliciously.

Method return values and/or error codes must accurately specify the object's state at an appropriate level of abstraction. Clients must be able to rely on the value for performing critical actions.

Noncompliant Code Example

The `updateNode()` method in this noncompliant code example modifies a node if it can find it in a linked list and does nothing if the node is not found.

```
public void updateNode(int id, int newValue) {
  Node current = root;
  while (current != null) {
    if (current.getId() == id) {
      current.setValue(newValue);
      break;
    }
    current = current.next;
  }
}
```

This method fails to indicate whether it modified any node. Consequently, a caller cannot determine that the method succeeded or failed silently.

Compliant Solution (Boolean)

This compliant solution returns the result of the operation as `true` if it modified a node and `false` if it did not.

```
public boolean updateNode(int id, int newValue) {
  Node current = root;
  while (current != null) {
    if (current.getId() == id) {
      current.setValue(newValue);
      return true; // Node successfully updated
    }
    current = current.next;
  }
  return false;
}
```

Compliant Solution (Exception)

This compliant solution returns the modified `Node` when one is found and throws a `NodeNotFoundException` when the node is not available in the list.

```
public Node updateNode(int id, int newValue)
    throws NodeNotFoundException {
  Node current = root;
  while (current != null) {
    if (current.getId() == id) {
      current.setValue(newValue);
      return current;
    }
    current = current.next;
  }
  throw new NodeNotFoundException();
}
```

Using exceptions to indicate failure can be a good design choice, but throwing exceptions is not always appropriate. In general, a method should throw an exception only when it is expected to succeed but an unrecoverable situation occurs or when it expects a method higher up in the call hierarchy to initiate recovery.

Compliant Solution (Null Return Value)

This compliant solution returns the updated `Node` so that the developer can simply check for a `null` return value if the operation fails.

```
public Node updateNode(int id, int newValue) {
  Node current = root;
  while (current != null) {
    if (current.getId() == id) {
      current.setValue(newValue);
      return current;
    }
    current = current.next;
  }
  return null;
}
```

A return value that might be null is an in-band error indicator, which is discussed more thoroughly in Guideline 52, "Avoid in-band error indicators." This design is

permitted but is considered inferior to other designs, such as those shown in the other compliant solutions in this guideline.

Applicability

Failure to provide appropriate feedback through a combination of return values, error codes, and exceptions can lead to inconsistent object state and unexpected program behavior.

Bibliography

[Bloch 2008] Item 59. Avoid unnecessary use of checked exceptions

[Long 2012] EXP00-J. Do not ignore values returned by methods

[Ware 2008] *Writing Secure Java Code*