Java Language Best Practices

This chapter describes Java language best practices. The topics include:

2.1 Avoid or Minimize Synchronization

Many performance studies have shown a high performance cost in using synchronization in Java. Improper synchronization can also cause a deadlock, which can result in complete loss of service because the system usually has to be shut down and restarted. But performance overhead cost is not a sufficient reason to avoid synchronization

completely. Failing to make sure your application is thread-safe in a multithreaded environment can cause data corruption, which can be much worse than losing performance. The following are some practices that you can consider to minimize the overhead:

- Synchronize Critical Sections Only
- Do Not Use the Same Lock on Objects That Are Not Manipulated Together
- Use Private Fields
- Use a Thread Safe Wrapper
- Use Immutable Objects
- Know Which Java Objects Already Have Synchronization Built-in
- Do Not Under-Synchronize

2.1.1 Synchronize Critical Sections Only

If only certain operations in the method must be synchronized, use a synchronized block with a mutex instead of synchronizing the entire method. For example:

```
private Object mutex = new Object();
    ...
    private void doSomething()
    {
     // perform tasks that do not require synchronicity
        ...
        synchronized (mutex)
            {
            ...
            }
        ...
    }
```

2.1.2 Do Not Use the Same Lock on Objects That Are Not Manipulated Together

Every Java object has a single lock associated with it. If unrelated operations within the class are forced to share the same lock, then they have to wait for the lock and must be executed one at a time. In this case, define a different mutex for each unrelated operation that requires synchronization.

Also, do not use the same lock to restrict access to objects that will never be shared by multiple threads. For example, using `Hashtables` to store objects that will never be accessed concurrently causes unnecessary synchronization overhead:

```
public class myClass
{
    private static myObject1 myObj1;
    private static mutex1 = new Object();
```

```
    private static myObject2 myObj2;
    private static mutex2 = new Object();
    ...
    public static void updateObject1()
        {
            synchronized(mutex1)
            {
            // update myObj1 ...
            }
        }
    public static void updateObject2()
    {
                synchronized(mutex2)
        {
        // update myObj2 ...
        }
    }
...
}
```

2.1.3 Use Private Fields

Making fields private protects them from unsynchronized access. Controlling their access means these fields need to be synchronized only in the class's critical sections when they are being modified.

2.1.4 Use a Thread Safe Wrapper

Provide a thread-safe wrapper on objects that are not thread-safe. This is the approach used by the collection interfaces in JDK 1.2.

2.1.5 Use Immutable Objects

An immutable object is one whose state cannot be changed once it is created. Since there is no method that can change the state of any of the object's instance variables once the object is created, there is no need to synchronize on any of the object's methods.

This approach works well for objects, which are small and contain simple data types. The disadvantage is that whenever you need a modified object, a new object has to be created. This may result in creating a lot of small and short-lived objects that have to be garbage collected. One alternative when using an immutable object is to also create a mutable wrapper similar to the thread-safe wrapper.

An example is the `String` and `StringBuffer` class in Java. The `String` class is immutable while its companion class `StringBuffer` is not. This is part of the reason why many Java performance books recommend using `StringBuffer` instead of string concatenation.

See Also:

Section 2.12, "Use Stringbuffer Instead of String Concatenation"

2.1.6 Know Which Java Objects Already Have Synchronization Built-in

Some Java objects (such as `Hashtable`, `Vector`, and `StringBuffer`) already have synchronization built into many of their APIs. They may not require additional synchronization.

2.1.7 Do Not Under-Synchronize

Some Java variables and operations are not atomic. If these variables or operations can be used by multiple threads, you must use synchronization to prevent data corruption. For example: (i) Java types long and double are comprised of eight bytes; any access to these fields must be synchronized. (ii) Operations such as ++ and -- must be synchronized because they represent a read and a write, not an atomic operation.

2.2 Monitor Synchronization

Java synchronization can cause a deadlock. The best way to avoid this problem is to avoid the use of Java synchronization. One of the most common uses of synchronization is to implement pooling of serially reusable objects. Often, you can simply add a serially reusable object to an existing pooled object. For example, you can add Java Database Connectivity (JDBC) and `Statement` object to the instance variables of a single thread model servlet, or you can use the Oracle JDBC connection pool rather than implement your own synchronized pool of connections and statements.

If you must use synchronization, you should either avoid deadlock, or detect it and break it. Both strategies require code changes. So, neither can be completely effective because some system code uses synchronization and cannot be changed by the application.

To prevent deadlock, simply number the objects that you must lock, and ensure that clients lock objects in the same order.

Proprietary JVM extensions may be available to help spot deadlocks without having to instrument code, but there are no standard JVM facilities for detecting deadlock.

2.3 Monitor and Fix Resource Leaks

One way to fix resource leaks is straightforward - a periodic restart. It provides good protection against slow resource leaks. But it is also important to spot applications that are draining resources too quickly, so that any software bugs can be fixed. Leaks that prevent continuous server operation for at least 24 hours must be fixed in the application code, not by application restart.

Common programming mistakes are:

- Not returning the resource to the pool (or not removing it from the pool) after handling an error.
- Relying on the garbage collector to invoke `finalize()` and free resources. Never rely on the garbage collector to manage any resource other than memory.
- Not discarding old object references which prevent recycling the memory occupied by the objects.

Monitoring resource usage should be a combination of code instrumentation and external monitoring utilities. With code instrumentation, calls to an application-provided interface, or calls to a system-provided interface like Oracle Dynamic Monitoring System (DMS), are inserted at key points in the application's resource usage lifecycle. Done correctly, this can give the most accurate picture of resource use. Unfortunately, the same programming errors that cause resource leaks are also likely to cause monitoring errors. That is, you may forget to release the resource, or forget to monitor the release of the resource.

Operating system commands like vmstat or ps in UNIX, provide process-level information such as the amount of memory allocated, the number and state of threads, or number of network connections. They can be used to detect a resource leak. Some commercially available development tools can also be used to find the leak.

In addition to compromising availability, resource leaks and overuse decrease performance.

See Also:

Section 2.8, "Avoid Common Errors That Can Result In Memory Leaks"

2.4 Always Use a Finally Clause In Each Method to Cleanup

In Java, it is impossible to leave the `try` or `catch` blocks (even with a `throw` or `return` statement) without executing the finally block. If for any reason the instance variables cannot be cleaned, throw a catch-all exception that should

cause the caller to discard its object reference to this now corrupt object. If, for any reason the static variables cannot be cleaned, throw an `InternalError` or equivalent that will ultimately result in restarting the now corrupt JVM.

2.5 Discard Objects That Throw Catch-All Exceptions

In many cases, these exceptions indicate that the internal state of the invoked object is corrupt, and that further invocations will also fail. Keep the object reference only if careful scrutiny of the exception shows it is benign, and further invocations on this object are likely to succeed.

Adopt a guilty unless proven innocent approach. For example, a `SQLException` thrown from an Oracle JDBC invocation could represent one of thousands of error conditions in the JDBC driver, the network, or the database server. Some of these errors (for example, subclass `SQLWarning`) are benign. Some `SQLExceptions` (for example, "ORA-3113: end of file on communication channel") definitely leave the JDBC object useless. Most `SQLExceptions` do not clearly specify what state the JDBC object is left in. The best approach is to enumerate the benign error codes that could occur frequently in your application and can definitely be retried, such as a unique key violation for user-supplied input data. If any other error code is found, discard the potentially corrupt object that threw the exception.

Discard all object references to the (potentially) corrupt object. Be sure to remove the corrupt object from all pools in order to prevent pools from being poisoned by corrupt objects. Do not invoke the corrupt object again - instantiate a brandnew object instead.

When you are sure that the corrupt objects have been discarded and that the catching object is not corrupt, throw a non catch-all exception so that the caller does not discard this object.

2.6 Design Transactions Usage Correctly

Transactions should not span client requests because this can tie up shared resources indefinitely.

Requests generally should not span more than one transaction, because a failure in mid-request could leave some transactions committed and others rolled back. If this requires application-level compensation to recover, then availability or data integrity may be compromised.

Transactions generally should not span more than one database, because distributed transactions lock shared resources longer, and failure recovery may require simultaneous availability and coordination of multiple databases.

Applications that require a single client request (for example, a confirm checkout request in a shopping cart application) to ultimately affect several databases (for example, credit card, fulfillment, shopping cart, and customer history databases) should perform the first step with one database, and in the same transaction queue a message in the first database addressed to the second database. The second database will perform the second step and queue the third step, and so on. This queued transaction chain will eventually complete automatically, or an administrator will see an undeliverable message and will have to manually compensate.

2.7 Put Business Logic In the Right Place

In general, you should not implement business logic in your client program. Instead, put validation and defaulting logic in your entity objects, and put client-callable methods in application modules, view objects, and view rows.

Working with application module methods allows the client program to encapsulate task-level custom code in a place that allows data-intensive operations to be done completely in the middle-tier without burdening the client.

Working with view object methods allows the client program to access the entire row collection for cross-row calculations and operations.

Working with view row methods allows the client program to operate on individual rows of data. There are three types of custom view row methods you may want to create:

- **Accessor methods**: The `oracle.jbo.Row` interface (which view rows implement) contains the methods `getAttribute()` and `setAttribute()`, but these methods are not typesafe. You can automatically generate custom typesafe accessors when you generate a custom view row class.
- **Delegators to entity methods**: By design, clients cannot directly access entity objects. If you want to expose an entity method to the client tier, you should create a delegator method in a view row.
- **Entity-independent calculations:** This is useful if the calculation uses attributes derived from multiple entity objects or from no entity objects.

2.8 Avoid Common Errors That Can Result In Memory Leaks

In Java, memory bugs often appear as performance problems, because memory leaks usually cause performance degradation. Because Java manages the memory automatically, developers do not control when and how garbage is collected. To avoid memory leaks, check your applications to make sure they:

- Release JDBC `ResultSet, Statement`, or connection.

Release failures here are usually in error conditions. Use a `finally` block to make sure these objects are released appropriately.

- Release instance or resource objects that are stored in static tables.

Perform clean up on serially reusable objects.

An example is appending error messages to a `Vector` defined in a serially reusable object. The application never cleaned the `Vector` before it was given to the next user. As the object was reused over and over again, error messages accumulated, causing a memory leak that was difficult to track down.

See Also:

Section 2.3, "Monitor and Fix Resource Leaks"

2.9 Avoid Creating Objects or Performing Operations That May Not Be Used

This mistake occurs most commonly in tracing or logging code that has a flag to turn the operation on or off during runtime. Some of this code goes to great lengths creating and formatting output without checking the flag first, creating many objects that are never used when the flag is off. This mistake can be quite expensive, because tracing and logging usually involves many `String` objects and operations to translate the message or even access to the database to retrieve the full text of the message. Large numbers of debug or trace statements in the code make matters worse.

2.10 Replace Hashtable and Vector With Hashmap, ArrayList, or LinkedList If Possible

The `Hashtable` and `Vector` classes in Java are very powerful, because they provide rich functions. Unfortunately, they can also be easily misused. Since these classes are heavily synchronized even for read operations, they can present some challenging problems in performance tuning. Hence, the recommendations are:

- Use an Array Instead of an ArrayList If the Size Can Be Fixed
- Use an ArrayList or LinkedList To Hold a List of Objects In a Particular Sequence
- Use HashMap or TreeMap To Hold Associated Pairs of Objects
- Replace Hashtable, Vector, and Stack
- Avoid Using String As the Hash Key (If Using JDK Prior to 1.2.2)

2.10.1 Use an Array Instead of an ArrayList If the Size Can Be Fixed

If you can determine the number of elements, use an `Array` instead of an `ArrayList`, because it is much faster. An `Array` also provides type checking, so there is no need to cast the result when looking up an object.

2.10.2 Use an ArrayList or LinkedList To Hold a List of Objects In a Particular Sequence

A `List` holds a sequence of objects in a particular order based on some numerical indexes. It will be automatically resized. In general, use an `ArrayList` if there are many random accesses. Use a `LinkedList` if there are many insertions and deletions in the middle of the list.

2.10.3 Use HashMap or TreeMap To Hold Associated Pairs of Objects

A Map is an associative array, which associates any one object with another object. Use a `HashMap` if the objects do not need to be stored in sorted order. Use `TreeMap` if the objects are to be in sorted order. Since a `TreeMap` has to keep the objects in order, it is usually slower than a `HashMap`.

2.10.4 Replace Hashtable, Vector, and Stack

- Replace a `Vector` with an `ArrayList` or a `LinkedList`.
- Replace a `Stack` with a `LinkedList`.
- Replace a `Hashtable` with a `HashMap` or a `TreeMap`.

`Vector`, `Stack`, and `Hashtable` are synchronized-views of `List` and Map. For example, you can create the equivalent of a `Hashtable` using:

```
private Map hashtable = Collections.synchronizedMap (new HashMap());
```

However, bear in mind that even though methods in these synchronized-views are thread-safe, iterations through these views are not safe. Therefore, they must be protected by a synchronized block.

2.10.5 Avoid Using String As the Hash Key (If Using JDK Prior to 1.2.2)

In Java's `HashMap` or `TreeMap` implementation, the `hashCode()` method on the key is invoked every time the key is accessed. If the hash key is a `String`, each access to the key will invoke the `hashCode()` and the `equals()` methods in the `String` class. Prior to JDK release 1.2.2, the `hashcode()` method in the `String` class did not cache the integer value of the `String` in an int variable; it had to scan each character in the `String` object each time. Such an operation can be very expensive. In fact, the longer the length of the `String`, the slower the `hashCode()` method.

2.11 Reuse Objects Instead of Creating New Ones If Possible

Object creation is an expensive operation in Java, with impact on both performance and memory consumption. The cost varies depending on the amount of initialization that needs to be performed when the object is to be created. Here are ways to minimize excess object creation and garbage collection overhead:

- Use a Pool to Share Resource Objects
- Recycle Objects
- Use Lazy Initialization to Defer Creating the Object Until You Need It.

2.11.1 Use a Pool to Share Resource Objects

Examples of resource objects are threads, JDBC connections, sockets, and complex user-defined objects. They are expensive to create, and pooling them reduces the overhead of repetitively creating and destroying them. On the down side, using a pool means you must implement the code to manage it and pay the overhead of synchronization when you get or remove objects from the pool. But the overall performance gain you get from using a pool to manage expensive resource objects outweighs that overhead.

However, be cautious on implementing a resource pool. The following mistakes in pool management are often observed:

- a resource object which should be used only serially is given to more than one user at the same time
- objects that are returned to the pool are not properly accounted for and are therefore not reused, wasting resources and causing a memory leak
- elements or object references kept in the pool are not reset or cleaned up properly before being given to the next user

These mistakes can have severe consequences including data corruption, memory leaks, a race condition, or even a security problem. Our advice in managing your pool is: keep your algorithm simple.

The J2EE section in this document includes examples showing how you can use Oracle's built-in JDBC connection caching and the servlet's `SingleThreadModel` to help manage a shared pool without implementing it yourself.

2.11.2 Recycle Objects

Recycling objects is similar to creating an object pool. But there is no need to manage it because the pool only has one object. This approach is most useful for relatively large container objects (such as `Vector` or `Hashtable`) that you want to use for holding some temporary data. Reusing these objects instead of creating new ones each time can avoid memory allocation and reduce garbage collection.

Similar to using a pool, you must take precautions to clear all the elements in any recycled object before you reuse it to avoid memory leak. The collection interfaces have the built-in `clear()` method that you can use. If you are building your object, you should remember to include a `reset()` or `clear()` method if necessary.

2.11.3 Use Lazy Initialization to Defer Creating the Object Until You Need It.

Defer creating an object until it is needed if the initialization of the object is expensive or if the object is needed only under some specific condition.

```
public class myClass
{
    private mySpecialObject  myObj;
    ...
    public mySpecialObject
             getSpecialObject()
    {
         if (myObj == null)
             myObj = new mySpecialObject();
         return myObj;
    }
    ...



}
```

2.12 Use Stringbuffer Instead of String Concatenation

The `String` class is the most commonly used class in Java. Especially in Web applications, it is used extensively to

generate and format HTML content.

`String` is designed to be immutable; in order to modify a `String`, you have to create a new `String` object. Therefore, string concatenation can result in creating many intermediate `String` objects before the final `String` can be constructed. `StringBuffer` is the mutable companion class of `String`; it allows you to modify the `String`. Therefore, `StringBuffer` is generally more efficient than `String` when concatenation is needed.

This section also features the following practices:

- Use StringBuffer Instead of String Concatenation If You Repeatedly Append to a String In Multiple Statements
- Use Either String or StringBuffer If the Concatenation Is Within One Statement
- Use StringBuffer Instead of String Concatenation If You Know the Size of the String

2.12.1 Use StringBuffer Instead of String Concatenation If You Repeatedly Append to a String In Multiple Statements

Using the "+=" operation on a `String` repeatedly is expensive.

For example:

```
String s = new
String();
     [do some work ...]
s += s1;
     [do some more work...]
s += s2;
```

Replace the above string concatenation with a StringBuffer:

```
StringBuffer strbuf = new StringBuffer();
     [do some work ...]
strbuf.append(s1);
     [so some more work ...]
strbuf.append(s2);
String s = strbuf.toString();
```

2.12.2 Use Either String or StringBuffer If the Concatenation Is Within One Statement

`String` and `StringBuffer` perform the same in some cases; so you do not need to use `StringBuffer` directly.

```
    String s = "a" + "b" + "c";
to
    String s = "abc";
```

Optimization is done automatically by the compiler.

- The Java2 compiler will automatically collapse the above.
- The Java2 compiler will also automatically convert the following:

```
        String s = s1 + s2;
    to
        String s = (new StringBuffer()).append(s1).append(s2).toString();
```

In these cases, there is no need to use `StringBuffer` directly.

2.12.3 Use StringBuffer Instead of String Concatenation If You Know the Size of the String

The default character buffer for `StringBuffer` is 16. When the buffer is full, a new one has to be re-allocated (usually at twice the size of the original one). The old buffer will be released after the content is copied to the new one. This constant reallocation can be avoided if the StringBuffer is created with a buffer size that is big enough to hold the `String`.

The following will be more efficient than using a `String` concatenation.

```
    String s = (new StringBuffer(1024)).
append(s1).append(s2).  toString();
```

will be faster than

```
    String s = s1 + s2;
```