Candle

Managing what matters most™

Java Coding Practices for Improved Application Performance

Lloyd Hagemo

Senior Director

Application Infrastructure Management Group

Candle Corporation

In the beginning, Java became the language of the Internet. It rode the hype of the World Wide Web, rapidly expanding in its scope and use. At first, developers didn't need to be concerned about performance. Their focus was on writing small applets that spun GIF or JPG images. Like the Web, Java continued to mature. Vendors developed higher-performing Java Virtual Machines (JVMs) and integrated development environments (IDEs) for coding and testing applications. Java became commonplace in most companies' application development departments, and major universities began teaching it as part of their computer science curricula. Performance was not as critical when Java was used in applets and stand-alone applications, but at center stage in the enterprise, high performance is key.

Sun Microsystems Inc. worked with several major software vendors within a consortium to create the Java 2 Enterprise Edition (J2EE) platform. A large group of vendors, including major players like BEA Systems Inc., Oracle Corp. and IBM Corp., adopted this standard. The appearance of industrial-strength implementations of the J2EE open standard in application servers and the availability of trained personnel helped drive the adoption of Java for enterprise application development. This Web language used for simple spinning images on Web pages moved to server-centric, transactional applications used to run a business. Today, Java is used to implement mission-critical applications. The performance of an application that services hundreds of concurrent users is critical.

In this environment, there are two types of performance considerations. The first is to improve the Java code, which can be done by applying some simple best practices. The second is focused on the usage of services provided by the J2EE application server containers. This article will focus on five key performance Java coding practices that can be used to improve the throughput of applications.

Avoiding Garbage Collection

The process of disposing of Java objects is called garbage collection (GC). GC is automatic in Java programs. When an object is no longer referenceable, it is available for GC. This occurs at the end of the block of code that references that object. GC is a low-priority JVM task that runs when there are CPU cycles available or the JVM runs short on memory. If the JVM is short on memory, GC will run continually until memory is available to run the system. This thread typically uses 5-15 percent of the CPU. There are two techniques that can be used to reduce GC. The first is to write applications that reuse existing objects; this eliminates the overhead of creating and destroying the object, thus reducing GC JVM overhead, but it does require extra work on the programmer because values will need to be reinitialized prior to reuse. The second technique is to use the appropriate objects that can meet the requirements. It is a well-known fact that string concatenation is an expensive operation in Java applications. This is because strings are immutable, meaning that the value of a string can never be changed. Consequently, every time two strings are concatenated, a new string must be created. The recommendation is to use a string buffer object instead. In the example below, several intermediate string objects are created as a result of the eight concatenations. Each of these intermediate string objects will need to be GC.

```
System.out.println(person.getLastName() + ", " + person.getFirstName() + " lives at " +
person.getAddress().getStreet1() + "; " + person.getAddress().getCity() + ", " + person.getAddress().getState() );
```

However, using a string buffer, the intermediate objects are not created requiring less GC while producing the same result.

```
StringBuffer buf = new StringBuffer;
buf.append(person.getLastName());
buf.append(", ");
buf.append(person.getFirstName());
buf.append(" lives at ");
buf.append(person.getAddress().getStreet1());
buf.append("; ");
buf.append(person.getAddress().getCity());
buf.append(", " );
buf.append(person.getAddress().getState());
System.out.println(buf.toString());
```

Loop Optimization

Java programs spend most of their time in loops. There are several techniques that can be used to optimize loops. Here are two examples that can be implemented easily to reduce a program's overhead.

Avoid using a method call as the termination criteria for a loop. For example: The code on the right will perform an average of 150 percent faster with the same results. This is because the length of the string is not calculated for each iteration through the loop. The method call to str.length() is avoided by setting the results to an integer prior to entering the loop.

```
public loop control()
{
  String str = "abcdefghijklmnopqurstuvwxyz";
  for (int j = 0; j < str.length(); j++)
  {
    // code doesn't change the length of the string.
  }
}
```

```
public loop control()
{
   String str = "abcdefghijklmnopqurstuvwxyz";
          int len = str.length();
   for (int j = 0; j < len; j++)
   {
     // code doesn't change the length of the string.
   }
}
```

The loop on the right can be improved even more by changing the loop to count backwards. The JVM is optimized to compare to integers between -1 and +5. So rewriting a loop to compare against 0 will produce faster loops. So for example, the loop on the right could be changed to use the following for loop instruction:

```
for (int j = len-1; j >= 0; j—)
```

There are other performance considerations when coding loops in Java. For example, once the processing requirement of a loop has been completed, use the break statement to terminate the loop. This saves the JVM from iterating through the loop doing the evaluations of the termination criteria. Using local variables within a loop requires less processing than instance variables. Local variables are defined within the scope of the method prior to the loop. Instance variables are part of a class object. Instance variables are looked up by the JVM, which can be expensive. For example, a class has an integer instance variable that needs to be set to an index value into an array. Within the method prior to the "for" loop, a local variable is defined. The local variable is used within the loop doing comparisons. After successfully locating the index, the loop is exited and the instance variable is set to the value of the local variable. In some cases, if the array is part of the object where the integer that needs to be set is located, it would be faster to create a local copy of the array and use that to determine the index within the loop.

Simple tests can be run by wrapping the code with System.currentTimeMillis() method calls to help you make the right choice. Performance testing may take a little extra time, but it is important to remember the life expectancy of an application is years. Therefore, investing in performance testing early in the life cycle drives significant return on investment throughout the application life cycle.

Data Structures

The data structure most easily modified for performance is collections. A collection is used as a generic term for any object that represents a set of items grouped together. There are three types of collections: sets, lists and maps. A set is a collection of objects that in the purest sense has no particular order. You can think of that as similar to having a collection of items stored in a shoebox. A list has the characteristics that objects are stored in a linear manner. An array is an example of a list where objects are stored into the list based upon the data structure's index. In most cases, order is not as important when using arrays. Maps involve pairs of objects, a key and the object itself. Objects are stored and retrieved using the keys. There are nine classes defined within the Java utility libraries that can be used to manage collections.

There are two classes that provide set collections: HashSet and TreeSet. HashSet is faster than TreeSet because TreeSet provides iteration of the keys in order. The implementations of sets are slower than most other collection objects and, therefore, careful consideration should be given prior to using this functionality.

There are three classes included in a map collection: HashMap, HashTable and TreeMap. A map is used to store data to minimize the need for searching when you want to retrieve an object. Both HashTables and HashMaps are fast, providing adequate performance. A TreeMap is slower than a HashMap for the same reason as outlined above regarding TreeSets and HastSets. The keys used in map collections have to be unique.

There are four classes included in a list: ArrayList, Vector, Stack and LinkedList. The ArrayList is the fastest of the list classes with Vector and Stack being about equal. Vector is slower than an ArrayList because of synchronization. Stack is implemented using the Vector class, but offers additional methods to push and pop entries.

Simple arrays provide the fastest data structure for storing data. The real advantage of Java is the libraries that have implemented advanced algorithms to make the management of data easier. These algorithms are optimized to bring better performance than could be implemented by the average application developer. And the code in these libraries is stable.

So what is the bottom line on collection performance? Proper sizing of the collection object is one of the most important considerations. For example, if the number of elements exceed the capacity of a HashTable, the program will end up with multiple nodes. Multiple nodes reduce a HashTable's efficiency. In addition, a larger hash code will ensure more even distribution within a HashTable, which improves performance. Vectors should be sized properly to avoid expansion. Adding and removing items from the end of a Vector improves performance because this avoids the shifting of existing elements. Unlike the Vector collection class, ArrayLists and HashMaps are not synchronized classes. When using multithreaded applications use ArrayLists and HashMaps to improve performance when synchronized access to the data is not a concern.

Synchronization

Java is designed to allow programmers to develop multithreaded applications. Threads allow applications to be designed so that independent operations can overlap. A lot of computers only have a single processor. So a Java program cannot execute more than one computation at any given instance that requires the CPU. You can overlap operations like disk input and output operations with processing activities. Disk IO is not dependent on continuous use of the processor. Multithreading is a powerful performance feature that can increase your application's throughput when used properly. Excessive synchronization defeats the performance improvements made possible by multithreading the application. Therefore, the recommendation is to design your applications to use the minimum amount of synchronization required for correct execution. The overhead of calling an unsynchronized method can be much smaller than that of calling a synchronized method.

There are two costs associated with synchronization. First, there is overhead associated with the management of locks by the JVM. This overhead is the work needed to monitor, test, acquire and release locks. For the JVM lock manager process, acquiring a lock is a synchronized activity to avoid two threads from obtaining the same lock. The second is the concept of synchronization itself. It defeats the purpose of building multithreaded applications that provide better performance through parallelism when threads end up waiting on synchronized processes.

Contention for locks can mean 10 times worse performance within an application. With a JIT compiler, that performance penalty can increase 50-100 times. The solution to this performance issue is having an understanding of which classes are appropriate to use within your application. For example, a Vector can be used effectively to store objects. You would naturally think that this object would be an effective technique to use to build lists of objects for reuse. Vector is an example of a synchronized class. This provides single-thread access to its data using locks. As explained in the section above, use HashMap and ArrayList classes when synchronized access to data is not required to improve performance.

There is one other consideration with synchronization. It is hard to predict when a class will be used by a multithreaded application. This could lead to program bugs that are hard to find. Within a multithreaded application, specifying the synchronized keyword on the method definition can synchronize individual methods. By using this standard Java feature, the programmer can write a wrapper class that subclasses nonsynchronized class definitions. This wrapper would synchronize the methods that could lead to problems in a multithreaded situation. Read-only methods would not be a candidate for synchronization, but add or update methods would be a consideration. This technique allows programmers to avoid general locking by using collection classes that are not synchronized except in the cases where the wrapper overrode the classes standard method definition.

Exception Handling

Exceptions are used in Java programs to signal an error or problem during the execution of a program. Exceptions are used extensively by the standard Java libraries. It is an indication that an operation within your application needs special attention. There are several different types of exceptions by coding errors, standard library methods, programmer self-defined and Java JVM. For example, a standard library method exception would be IndexOutOfBoundsException, thrown when a program uses an index outside the bounds of the object for an array. For almost all the exceptions represented by subclasses of the exception class, you must include code in your program to deal with them. A try-catch block is used to handle the processing of exceptions. Exceptions must be handled by the calling method in a Java catch-code block. That calling method has an option to register that it will throw an exception in its method definition to pass the error to the next higher point in the calling sequence. Your program will not compile if exceptions are not handled.

A simple performance improvement can be achieved by placing the try-catch block outside any loops. On some JVMs, this can amount to as much as a 10-percent increase in performance. In the figure below, example one will run faster on most JVMs than example two. In example one, the try-catch block is outside the "for" loop. In this example, an exception is not thrown. If the evaluation in the "for" loop expression was changed to j>= -1, the ArthmeticException would be thrown.

```
//Example One:
public static main(String[] args)
{
 int i = 12;
 try
 {
  for(int j = 3; j >=  1; j—)
  {
    System.out.println(i/j);
  }
 }
 catch(ArithmeticException e)
 {
  System.err.println(e);
 }
}
```

```
// Example two:
public static main(String[] args)
{
 int i = 12;
for(int j = 3; j >=  1; j—)
 {
        try
 {
      System.out.println(i/j);
   }
   catch(ArithmeticException e)
   {
      System.err.println(e);
      }
      }
    }
```

Whenever an exception is thrown, the JVM must execute several hundred lines of code to handle the exception. A lot of this overhead is due to getting a snapshot of, and unwinding, the stack when the exception occurs. So exceptions should be used only for extreme conditions.

There are times when you need to have an exception thrown despite the overhead associated with exception processing. This is the case where you are going to throw a self-defined exception. The overhead can be reduced 50-100 times by defining the exception object and reusing it. Below is an example where we define the exception object in the init() method. Then when the exception is thrown it is reused.

```
// Reuse of Exception Object
public static Exception REUSABLE_EX = new Exception();
public void method1() throws EXCEPTION
{
  if (I == 1)
    throw REUSABLE_EX;
}
```

The alternative would be to code "throw new Exception(); // 50 –100 times slower" – replacing the previous throw instruction.

Conclusion

Java is a powerful language that has moved to center stage as an enterprise development tool. It is being used today to build business critical applications. Generally available application servers from major vendors support the wide range of standards that allow customers to quickly build secure server-side transactional business applications. Performance is a key consideration when Java applications are being used as customer-facing applications that drive your business. Proper coding is certainly one step toward providing applications that meet performance requirements. There are myriad other considerations that must be taken into account. These include products that provide unit testing capabilities, profilers, automated test tools and monitoring tools. In all cases, the test results need to match the performance requirements for the application. Monitors are designed to help you measure that performance from several perspectives, including the operating systems, Web application servers and databases. Server-side Java development has made it necessary for a developer to understand the characteristics of the language's performance to support mission-critical and customer-facing business applications. Tools can validate and document your skills to deliver high-performance mission-critical results.

Candle®

World Headquarters – Candle Corporation, 100 North Sepulveda Boulevard, El Segundo, CA 90245 (800) 97CANDLE
Europe – Camberley, U.K. (44) 1276 41 47 00; fax (44) 1276 41 47 77
Asia Pacific – Singapore (65) 220-5092; fax (65) 226-3579

For a listing of our offices and business partners worldwide, visit our Web site: ***www.candle.com***
Australia - Austria - Belgium - Brazil - Canada - Denmark - Finland - France - Germany - Hong Kong - Italy - Japan - Korea - Malaysia - Netherlands - Norway - Singapore - Sweden - Switzerland - Taiwan - United Kingdom - United States