```
    byte[ ] one_byte = new byte[1];
    public SockOutStreamLogger(Socket so, OutputStream o){out = o; s = so;}
    public void write(int b) throws IOException {
      out.write(b);
      one_byte[0] = (byte) b;
      SockStreamLogger.written(s, 1, one_byte, 0);
    }
    public void write(byte b[ ]) throws IOException {
      out.write(b);
      SockStreamLogger.written(s, b.length, b, 0);
    }
    public void write(byte b[ ], int off, int len) throws IOException {
      out.write(b, off, len);
      SockStreamLogger.written(s, len, b, off);
    }
    public void flush( ) throws IOException {out.flush( );}
    public void close( ) throws IOException {out.close( );}
}
```

Performance Checklist

- Use system- and network-level monitoring utilities to assist when measuring performance.
- Run tests on unloaded systems with the test running in the foreground.
  - Use `System.currentTimeMillis( )` to get timestamps if you need to determine absolute times. Never use the timings obtained from a profiler as absolute times.
  - Account for performance effects of any caches.
- Get better profiling tools. The better your tools, the faster and more effective your tuning.
  - Pinpoint the bottlenecks in the application: with profilers, by instrumenting code (putting in explicit timing statements), and by analyzing the code.
  - Target the top five to ten methods, and choose the quickest to fix.
  - Speed up the bottleneck methods that can be fixed the quickest.
  - Improve the method directly when the method takes a significant percentage of time and is not called too often.
  - Reduce the number of times a method is called when the method takes a significant percentage of time and is also called frequently.
- Use an object-creation profiler together with garbage-collection statistics to determine which objects are created in large numbers and which large objects are created.
  - See if the garbage collector executes more often than you expect.
  - Determine the percentage of time spent in garbage collection and reduce that if over 15% (target 5% ideally).

Bron: Java Performance Tuning, Jack Shirazi, ed. 2003 (2nd ed)
ISBN 0-596-00377-3

  - Use the `Runtime.totalMemory()` and `Runtime.freeMemory()` methods to monitor gross memory usage.
- Check whether your communication layer has built-in tracing features.
  - Check whether your communication layer supports the addition of customized layers.
- Identify the number of incoming and outgoing transfers and the amounts of data transferred in distributed applications.

many small or negligible hits, thus spreading the load over time. This is essentially the same technique as for preallocation of objects (see the previous section).

It is true that many of these kinds of situations should be anticipated at the design stage, in which case you could build lazy initialization into the application from the beginning. But this is quite an easy change to make (usually affecting just the accessors of a few classes), and so there is usually little reason to over-engineer the application prior to tuning.

Performance Checklist

Most of these suggestions apply only after a bottleneck has been identified:

- Establish whether you have a memory problem.
- Reduce the number of temporary objects being used, especially in loops.
  - Avoid creating temporary objects within frequently called methods.
  - Presize collection objects.
  - Reuse objects where possible.
  - Empty collection objects before reusing them. (Do not shrink them unless they are very large.)
  - Use custom conversion methods for converting between data types (especially strings and streams) to reduce the number of temporary objects.
  - Define methods that accept reusable objects to be filled in with data, rather than methods that return objects holding that data. (Or you can return immutable objects.)
  - Canonicalize objects wherever possible. Compare canonicalized objects by identity.
  - Create only the number of objects a class logically needs (if that is a small number of objects).
  - Replace strings and other objects with integer constants. Compare these integers by identity.
  - Use primitive data types instead of objects as instance variables.
  - Avoid creating an object that is only for accessing a method.
  - Flatten objects to reduce the number of nested objects.
  - Preallocate storage for large collections of objects by mapping the instance variables into multiple arrays.
  - Use `StringBuffer` rather than the string concatenation operator (+).
  - Use methods that alter objects directly without making copies.
  - Create or use specific classes that handle primitive data types rather than wrapping the primitive data types.

- Consider using a `ThreadLocal` to provide threaded access to singletons with state.
- Use the `final` modifier on instance-variable definitions to create immutable internally accessible objects.
- Use `WeakReferences` to hold elements in large canonical lookup tables. (Use `SoftReferences` for cache elements.)
- Reduce object-creation bottlenecks by targeting the object-creation process.
  - Keep constructors simple and inheritance hierarchies shallow.
  - Avoid initializing instance variables more than once.
  - Use the `clone( )` method to avoid calling any constructors.
  - Clone arrays if that makes their creation faster.
  - Create copies of simple arrays faster by initializing them; create copies of complex arrays faster by cloning them.
- Eliminate object-creation bottlenecks by moving object creation to an alternative time.
  - Create objects early, when there is spare time in the application, and hold those objects until required.
  - Use lazy initialization when there are objects or variables that may never be used, or when you need to distribute the load of creating objects.
  - Use lazy initialization only when there is a defined merit in the design, or when identifying a bottleneck that is alleviated using lazy initialization.

pointless to use it exactly as defined here because object-creation overhead means that using the `PartialSorter` is twice as slow as using the `String.compareTo()` directly. But customizing the `PartialSorter.compare()` method for any particular locale is a reasonable task: remember, we are interested only in a simple algorithm that handles a partial sort, not the full intricacies of completely accurate locale-specific comparison.

Generally, you cannot expect to support internationalized strings and retain the performance of simple one-byte-per-character strings. But, as shown here, you can certainly improve the performance.

Performance Checklist

Most of these suggestions apply only after a bottleneck has been identified:

- Logically partition your strings into those that require internationalization support (i.e., text) and those that don't.
  - Avoid internationalization where the `Strings` never require it.
- Avoid using the `StreamTokenizer`.
- Regular expressions provide acceptable performance compared with using `String` searching methods and `String` character iteration tokenizing techniques.
- Create and optimize your own framework to convert objects and primitives to and from strings.
- Use efficient methods of `String` that do not copy the characters of the string, e.g., `String.substring()`.
  - Avoid using inefficient methods of `String` that copy the characters of the string, e.g., `String.toUppercase()` and `String.toLowercase()`.
  - Use the string concatenation operator to create `Strings` at compile time.
  - Use `StringBuffers` to create `Strings` at runtime.
  - Specify when the underlying `char` array is copied when reusing `StringBuffers`.
- Improve access to the underlying `String` `char` array by copying the `chars` into your own array.
  - Manipulate characters in `char` arrays rather than using `String` and `StringBuffer` manipulation.
  - Reuse `char` arrays.
- Optimize the string comparison and search algorithm for the data being compared and searched.
  - Compare strings by identity.
  - Convert a comparison task to a (hash) table lookup.

— Handle case-insensitive comparisons differently from case-sensitive comparisons.

— Apply the standard performance optimization for case-insensitive access (maintaining a second collection with all strings uppercased).

— Use `java.text.CollationKeys` rather than a `java.text.Collator` object to sort international strings.

— Use `String.compareTo( )` for string comparison where internationalization is unnecessary.

— Partially sort (international) strings using a simple comparison algorithm before using the full (internationalized) comparison.

example, when parsing a `String` object, it is common not to pass the length of the string to methods because each method can get the length using the `String.length( )` method. But parsing tends to be intensive and recursive, with lots of method calls. Most of those methods need to know the length of the string. Although you can eliminate multiple calls within one method by assigning the length to a temporary variable, you cannot do that when many methods need that length. Passing the string length as a parameter is almost certainly cheaper than repeated calls to `String.length( )`.

Similarly, you typically access the elements of the string one at a time using `String.charAt( )`. But again, it is better for performance purposes to copy the `String` object into a `char` array and pass this array through your methods (see Chapter 5). To provide a possible performance boost, try passing extra values and arrays to isolated groups of methods. As usual, you should do this only when a bottleneck has been identified, not throughout an implementation.

Finally, you can reduce the number of objects used by an application by passing an object into a method, which then fills in the object's fields. This is almost always more efficient than creating new objects within the method. See "Reusable Parameters" in Chapter 4 for a more detailed explanation of this technique.

Performance Checklist

Most of these suggestions apply only after a bottleneck has been identified:

- Include all error-condition checking in blocks guarded by `if` statements.
- Avoid throwing exceptions in the normal code path of your application.
- Investigate whether a `try-catch` in the bottleneck imposes any extra cost.
- Use `instanceof` instead of making speculative class casts in a `try-catch` block.
- Consider throwing exceptions without generating a stack trace by reusing a previously created instance.
- Include any exceptions generated during the normal flow of the program when running performance tests.
- Assertions add overhead even when disabled, though an optimizing JIT compiler can eliminate the overhead (only HotSpot server mode succeeded in 1.4.0).
- Beware of adding assertions to quick, frequently called methods.
- Minimize casting.
- Avoid casts by creating and using type-specific collection classes.
- Use temporary variables of the cast type, instead of repeatedly casting.
- Type variables as precisely as possible.
- Use local variables rather than instance or static variables for faster manipulation.

- Use temporary variables to manipulate instance variables, static variables, and array elements.
- Use ints in preference to any other data type.
- Avoid long and double instance or static variables.
- Use primitive data types instead of objects for temporary variables.
- Consider accessing instance variables directly rather than through accessor methods. (But note that this breaks encapsulation.)
- Add extra method parameters when that would allow a method to avoid additional method calls.

the extra variable holding the stack becomes empty. Otherwise, instead of the recursive call, the directory is added to the stack.

In the cases of these particular search methods, the time-measurement comparison shows that the iterative method actually takes 5% longer than the recursive method. This is due to the iterative method having the overhead of the extra stack object to manipulate, whereas filesystems are generally not particularly deep (the ones I tested on were not), so the recursive algorithm is not particularly inefficient. This illustrates that a recursive method is not always worse than an iterative one.

Note that the methods here were chosen for illustration, using an easily understood problem that could be managed iteratively and recursively. Since the I/O is actually the limiting factor for these methods, there would not be much point in actually making the optimization shown.

For this example, I eliminated the I/O overhead, as it would have swamped the times and made it difficult to determine the difference between the two implementations. To do this, I mapped the filesystem into memory using a simple replacement of the `java.io.File` class. This stored a snapshot of the filesystem in a hash table. (Actually, only the full pathnames of directories as keys, and their associated string array list of files as values, need be stored.)

This kind of trick—replacing classes with another implementation to eliminate extraneous overhead—is quite useful when you need to identify exactly where times are going.

Performance Checklist

Most of these suggestions apply only after a bottleneck has been identified:

- Make the loop do as little as possible.
  - Remove from the loop any execution code that does not need to be executed on each pass.
  - Move any code that is repeatedly executed with the same result, and assign that code to a temporary variable before the loop ("code motion").
  - Avoid method calls in loops when possible, even if this requires rewriting or inlining.
  - Multiple access or update to the same array element should be done on a temporary variable and assigned back to the array element when the loop is finished.
  - Avoid using a method call in the loop termination test.
  - Use `int` data types preferentially, especially for the loop variable.
  - Use `System.arraycopy( )` for copying arrays.
  - Try to use the fastest tests in loops.

- Convert equality comparisons to identity comparisons when possible.
- Phrase multiple boolean tests in one expression so that they "short circuit" as soon as possible.
- Eliminate unneeded temporary variables from loops.
- Try unrolling the loop to various degrees to see if this improves speed.

- Rewrite any `switch` statements to use a contiguous range of case values.
- Identify whether a recursive method can be made faster.
  - Convert recursive methods to use iteration instead.
  - Convert recursive methods to use tail recursion.
  - Try caching recursively calculated values to reduce the depth of recursion.
  - Use temporary variables in place of passed parameters to convert a recursive method using a single search path into an iterative method.
  - Use temporary stacks in place of passed parameters to convert a recursive method using multiple search paths into an iterative method.