Thirty Ways to Improve the Performance of Your Java™ Programs

Glen McCluskey

E-mail: jperf@glenmccl.com

Version 1.0, October 1999

The author would like to thank Steve Buroff, Craig Hondo, John Spicer, and Clay Wilson for assistance with proofreading.

Contents

1 Introduction

This paper presents 30 ways to improve the performance of your Java™ applications. These techniques focus on Java language and library features. Performance is defined to include both speed and space issues, that is, how to make your programs run faster, while using less memory and disk space.

The paper raises a variety of performance issues, and gives some hard numbers about how specific performance improvements work out. It should be noted up front that there's no way to present totally definitive advice on performance, because various applications have different performance characteristics and bottlenecks, and because performance varies across different hardware, operating systems, and Java development tools such as compilers and virtual machines. The Java programming language is still evolving, and its performance continues to improve. The ultimate aim of the paper is to promote awareness of Java performance issues, so that you can make appropriate design and implementation choices for specific applications.

1.1 When to Worry About Performance

Performance is not the only consideration in developing applications. Issues like code quality, maintainability, and readability are of equal or greater importance. The techniques found in the paper are not intended to be applied in isolation, simply because they exist. Some of the techniques are "tricky", leading to more obscure code, and should be applied only when necessary.

For example, consider the use of the string concatenation operator (`+`). It's natural to say:

```
String s = s1 + s2 + s3 + "abc" + s4 + "def";
```

to concatenate a set of strings. Most of the time this is the right thing to do, though sometimes you will want to apply other approaches described below (§4.1), approaches that are more efficient at the expense of naturalness of expression.

1.2 Performance Issues Not Covered in This Paper

This paper describes a set of techniques, rooted in the Java language and libraries. There are other areas of performance mentioned only in passing.

The first of these is *algorithm performance*. If your application contains fundamentally slow algorithms in it, these techniques may not help you. For example, suppose that you use an `N*N` sort algorithm instead of a much more efficient `N*log(N)` one. It's possible that such an algorithm will dominate the running time of your application, and there is nothing in the paper that will help you in such a case.

The other area is *architecture*. Sometimes poor performance is literally "built in" to an application, making it very difficult to do any useful performance improvement by tuning. For example, suppose that you make heavy use of the object-oriented paradigm, such that you have many layers of superclasses and subclasses. Moreover, suppose your application is one that processes bytes or characters one at a time, and each byte is processed through all these layers of classes and methods. In such a case, adequate performance may be impossible to attain, short of an application re-design. It's important to keep performance in mind when designing your applications.

1.3 Just-in-Time Compilers and Java Virtual Machines

Newer versions of Java tools offer a variety of approaches to improve performance, compared to the original scheme of bytecode interpretation. These approaches involve compiling bytecodes at run time into

machine code, streamlined versions of the Java Virtual Machine, and so on. Some of the areas most affected by these performance enhancements include:

- Method inlining.
- Adaptive inlining, where methods are inlined based on their use.
- Garbage collection.
- Method locking (`synchronized` methods).
- Compiling the Java language directly into native code.

These tools are quite important and worth examining to see if they will speed up your applications. In one case, an application that sorts a large list of numbers, there is a 40-1 speed difference between bytecode interpretation and just-in-time compilation. A large speedup like this is much more than you should typically expect from any of the techniques found in this paper.

1.4 Environment and Tools Used in Code Examples

The examples in this paper were developed against the Java Development Kit 1.2.2 (JDK™) from Sun, that implements the Java 2 version of the language and libraries. The JDK was run on a Windows NT 4.0 SP5 system, a 300 MHz Pentium with 128 MB of memory.

Compilation of examples was done by saying:

```
$ javac prog.java
```

and examples run with:

```
$ java –Djava.compiler=NONE prog
```

In other words, no optimization was used during compilation, and no just-in-time or other special compiler technology was used. This was a deliberate choice, in that the purpose of the paper is to compare different techniques against each other, rather than to come up with absolute measurements of time, and various compiler optimizations will tend to make some of the differences disappear.

It's important to note that your experience with some of the examples in the paper may vary widely from the results reported here. For example, the section of the paper dealing with strings (§4) discusses the overhead incurred by repeatedly calling the `length()` and `charAt()` methods of `String`. If you have an optimizing compiler available, these methods are likely to be inlined, and you will get different results.

1.5 How Examples Were Timed

The examples in the paper use a special `Timer` class, defined as follows:

```
// Timer class

public class Timer {
        long t;

        // constructor

        public Timer() {
                reset();
        }

        // reset timer

        public void reset() {
                t = System.currentTimeMillis();
        }

        // return elapsed time

        public long elapsed() {
                return System.currentTimeMillis() - t;
        }

        // print explanatory string and elapsed time

        public void print(String s) {
                System.out.println(s + ": " + elapsed());
        }
}
```

Calling the constructor or the `reset()` method records the current system time as a base time. Then `elapsed()` is called to get the difference between the current time and the base that was recorded, and `print()` is used to display the difference along with an explanatory string.

Typical usage of the class is:

```
Timer t = new Timer();

... do something ...

t.print("time was");
```

This results in output like:

```
time was: 784
```

where 784 is a time in milliseconds.

1.6 Performance Analysis Tools

There are a variety of Java performance analysis tools available today. One that comes with JDK 1.2.2 is invoked by saying:

```
$ java –Xrunhprof:cpu=times prog
```

with the results written to a file `java.hprof.txt`. This tool provides information on the time spent in each method in the application.

Another tool is:

```
$ javap –c prog
```

used to display Java Virtual Machine bytecodes. Using `javap` you can check for the presence of particular constructs in your program, such as generated constructors (§2.1). As an example of `javap` output, for the program:

```
public class hello {
        public static void main(String args[]) {
                System.out.println("Hello world");
        }
}
```

`javap –c` output is:

```
Compiled from hello.java
public class hello extends java.lang.Object {
    public hello();
    public static void main(java.lang.String[]);
}

Method hello()
   0 aload_0
   1 invokespecial #6 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 getstatic #7 <Field java.io.PrintStream out>
   3 ldc #1 <String "Hello world">
   5 invokevirtual #8 <Method void println(java.lang.String)>
   8 return
```

1.7 Web Site

There is a support web site associated with this paper. It can be found at:

```
http://www.glenmccl.com/jperf/index.htm
```

Any corrections to the paper will be listed here.

If you have any suggestions or comments on the paper, please direct them to:

```
jperf@glenmccl.com
```

You can use the same address if you are interested in setting up a consulting arrangement with the author.

2 Classes

This section discusses some performance issues with constructor calls and class initialization.

2.1 Default Constructors

Suppose that you have an innocent-looking bit of code such as this:

```
// default constructors

public class cls_default {
        public static void main(String args[]) {
                cls_default x = new cls_default();
        }
}
```

When `new` is called, to create an instance of the `cls_default` class, there are no calls to constructors, right? Actually, there are *two* constructor calls, as illustrated by the output of `javap -c` run on this class:

```
Method cls_default()
   0 aload_0
   1 invokespecial #4 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 new #1 <Class cls_default>
   3 dup
   4 invokespecial #3 <Method cls_default()>
   7 astore_1
   8 return
```

The `invokespecial` calls are Java Virtual Machine bytecodes used to call constructors.

The Java language has the notion of a *default constructor*, a constructor automatically generated by a compiler if no constructor is defined in a class. A generated constructor invokes the constructor of the class's superclass. For example, the declaration above that reads:

```
public class cls_default
```

actually means:

```
public class cls_default extends java.lang.Object
```

In other words, `Object` is the superclass of any class you define that does not otherwise have a superclass specified, and its constructor will be invoked by a generated constructor in your class.

So a constructor is generated that invokes the constructor in `java.lang.Object`. `java.lang.Object` is a special case, in that it has no constructor declared, and no superclass. The generated constructor for `java.lang.Object` simply has an empty body.[1] Constructor calls matter because of method call overhead (§3.1).

[1] See section 8.6.7 in the *Java™ Language Specification* for further details on default constructors.

2.2 Constructor Hierarchies

We can generalize the idea in the previous section. Suppose that you have a class hierarchy:

```
class A {...}
class B extends A {...}
class C extends B {...}
```

If you create a new instance of class `C`, then a chain of constructors are called, clear back to the one in `java.lang.Object`. This is the way it's supposed to work, but at some point the expense can mount up. This example illustrates the importance of method inlining (§3.1), and tools that help in such inlining (§1.3).

2.3 Class and Instance Initialization

When an instance of a class is created using `new`, initialization of the class's instance variables (variables unique to each instance) must be done. By contrast, class variables (those declared `static`, and shared across instances) need only be initialized once, conceptually at program invocation time[2]. The difference between these types of initialization is quite important, as this example illustrates:

```
// class initialization

public class cls_init1 {
        static class Data {
                private int month;
                private String name;
                Data(int i, String s) {
                        month = i;
                        name = s;
                }
        }
        Data months[] = {
                new Data(1, "January"),
                new Data(2, "February"),
                new Data(3, "March"),
                new Data(4, "April"),
                new Data(5, "May"),
                new Data(6, "June")
        };
        public static void main(String args[]) {
                final int N = 250000;
                cls_init1 x;
                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        x = new cls_init1();
                t.print("data declared non-static");
        }
}
```

This example takes 4627 units of time to run. However, if we look closely at this class, there is a potential inefficiency. The month number/name data found in `months[]` is an instance variable of the class, that

[2] See 12.4.1 in the *Java™ Language Specification* for exact details of class initialization.

is, a copy of the data is found in every instance of the class. Structuring the data in this way doesn't make sense, in that the number/name data never changes, and is the same across all class instances.

So we can change the program slightly, to turn the number/name data into a class variable, with a single copy across all instances:

```
// class initialization

public class cls_init2 {
        static class Data {
                private int month;
                private String name;
                Data(int i, String s) {
                        month = i;
                        name = s;
                }
        }
        static Data months[] = {
                new Data(1, "January"),
                new Data(2, "February"),
                new Data(3, "March"),
                new Data(4, "April"),
                new Data(5, "May"),
                new Data(6, "June")
        };
        public static void main(String args[]) {
                final int N = 250000;
                cls_init2 x;
                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        x = new cls_init2();
                t.print("data declared static");
        }
}
```

This program requires 421 units of time to run, a saving of 10-1 over the first approach. Moreover, it saves a lot of space per class instance as well. As a general rule, it's worth "factoring out" methods that do not operate on unique data in a class instance, and variables that are not unique to an instance, and making both methods and variables `static`, that is, shared across all instances.

2.4 Recycling Class Instances

Suppose that you are implementing some type of a linked list structure, and that nodes of the structure have the form:

```
class Node {
   Object data; // data
   Node link;   // link to next node in the list
}
```

and you are worried about the costs of allocating and garbage collecting large numbers of these nodes.

One alternative to `new` and garbage collection is to keep a free list of nodes, and check the list before you call `new` to allocate a node. For example, you might have a class variable `freelist` used for this purpose. When you free up a node referenced by a variable `noderef`, you could say:

```
noderef.link = freelist;
freelist = noderef;
```

that is, add the freed node to the head of the free list. When you want to allocate a node, you check `freelist`, and if it's not `null`, you can obtain a node by saying:

```
noderef = freelist;
freelist = noderef.link;
```

For this scheme to work, you need to update `freelist` in a thread-safe manner, by using `synchronized` methods or statements.

You need to be careful with this technique, as it tends to work against the whole idea of constructors, and against a cleanly defined approach to initializing objects. This approach makes most sense when you have some type of an internal utility class, and you want to manage large numbers of instances of this class.

3 Methods

There is an intrinsic cost associated with calling Java methods. These costs involve actual transfer of control to the method, parameter passing, return value passing, and establishment of the called method's stack frame where local variables are stored. Such costs show up in other languages as well. In this section we will look at a few of the performance issues with methods.

3.1 Inlining

Perhaps the most effective way to deal with method call overhead is method *inlining*, either by a compiler doing it automatically, or doing it yourself manually. Inlining is done by expanding the inlined method's code in the code that calls the method. Consider this example:

```
// method inlining

public class meth_inline {
        public static int min(int a, int b) {
                return (a < b ? a : b);
        }
        public static void main(String args[]) {
                final int N = 10000000;
                int a = 5;
                int b = 17;
                int c;

                // call a method

                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        c = min(a, b);
                t.print("method called");

                // inline the same method

                t.reset();
                for (int i = 1; i <= N; i++)
                        c = (a < b ? a : b);
                t.print("method inlined");
        }
}
```

The first case takes 6470 units of time, and the second 3364, nearly twice as fast. Note in this particular example that the JDK 1.2.2 just-in-time compiler will by default inline the `min()` method. We've turned off just-in-time compilation for the timings in this paper.

There are several ways that compilers can perform automatic inlining. One way is to expand the called method inline in the caller, which improves speed at the expense of code space. Another approach is more dynamic, where methods are inlined in a running program.

Note that it's possible to make methods more attractive for inlining, by keeping them short and simple. For example, a method that simply returns the value of a private field is a prime candidate for inlining.

3.2 Final Methods

One way you can help a compiler with inlining is to declare methods as `final`, that is, declaring that no subclass method overrides the method:

```
public final void f() {...}
```

The same technique is used with whole classes, indicating that the class cannot be subclassed, and that all its methods are implicitly `final`:

```
public final class A {...}
```

The idea behind use of `final` is this. Suppose that I have a reference to a `final` class `A`:

```
A aref = new A();
```

and I call a method `f()`:

```
aref.f();
```

At this point I know that I really am calling `A`'s method `f()`, and not some `f()` in a subclass of `A` (because `A` has no subclasses). In other words, use of `final` provides hints about what are typically called *virtual functions*, that is, methods or functions that are dispatched at run time based on the type of the object that they are called for. It's harder to inline a virtual method, since the actual method to be called cannot be determined at compile time.

3.3 Synchronized Methods

Synchronized methods are used in thread programming, where you want to grant a method exclusive access to a given object instance, for example so that the method can perform data structure updates without interference from other threads. Such methods are slower than non-synchronized ones, because of the overhead associated with obtaining a lock on the method's object. Consider this code:

```
// synchronized methods
public class meth_sync {
        public void meth1() {}
        public synchronized void meth2() {}
        public static void main(String args[]) {
                final int N = 1000000;
                meth_sync x = new meth_sync();

                // non-synchronized
                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        x.meth1();
                t.print("non-synchronized");

                // synchronized
                t.reset();
                for (int i = 1; i <= N; i++)
                        x.meth2();
                t.print("synchronized");
        }
}
```

The synchronized call takes about twice as long as the non-synchronized one. This difference is important in some contexts, for example in the collection classes (§6.2). Collection classes like `ArrayList` are not themselves thread-safe, but they have mechanisms whereby you can add a thread-safe wrapper on top of a collection class instance. This approach is useful in designing your own classes.

3.4 Inner Classes

An interesting situation with method call overhead arises when you use inner classes. The inner class specification says that a private method of a class `A` can be used by a class `B`, if `A` encloses `B`. That is, if `B` is a class defined within `A`, it can call `A`'s private methods. Here is an example:

```
// methods and inner classes

public class meth_inner {
        private void f() {}
        class A {
                A() {
                        f();
                }
        }
        public meth_inner() {
                A a = new A();
        }
        public static void main(String args[]) {
                meth_inner x = new meth_inner();
        }
}
```

`A` is an inner class defined within `meth_inner`, and `A`'s constructor calls a private method `f()` defined in `meth_inner`. Because the Java Virtual Machine has restrictions on calling private members from outside of their class, a special access method is generated by the compiler and added internally to the `meth_inner` class. This method has a name `access$0()`, and it in turns calls `f()`. In other words, a method is generated that can be called from outside of `meth_inner`, and grant access to a private method of `meth_inner`. So when `f()` is called above, a generated method `access$0()` is called, and it in turn calls `f()`.

If you use the JDK utility program (§1.6) that disassembles `.class` files, by saying:

```
$ javap –c meth_inner
```

the output includes the following method definition:

```
Method void access$0(meth_inner)
   0 aload_0
   1 invokespecial #7 <Method void f()>
   4 return
```

This is the body of the generated method.

You can avoid this overhead by avoiding use of private members in a class, assuming the class has inner classes that use those members. Obviously, however, there may be other reasons why private members are more important than the overhead associated with these generated methods.

4 Strings

Strings are a widely used data type in the Java language. Java strings are represented as objects of type `String`, and store sequences of 16-bit Unicode characters, along with the current string length.

4.1 Strings are Immutable

Perhaps the most important point about Java strings relative to performance is that strings are *immutable*, that is, they never change after creation. For example, in this sequence:

```
String str = "testing";
str = str + "abc";
```

the string "testing", once created, does not change, but a *reference* to the string may change. The string reference in `str` originally points to "testing", but then is changed to point to a new string formed by concatenating `str` and "abc". The above sequence is implemented internally using code like:

```
String str = "testing";
StringBuffer tmp = new StringBuffer(str);
tmp.append("abc");
str = tmp.toString();
```

In other words, the two strings to be concatenated are copied to a temporary string buffer, then copied back. Such copying is quite expensive. So a fundamental performance rule to remember with strings is to use `StringBuffer` objects explicitly if you're building up a string. String concatenation operators like `+` and `+=` are fine for casual use, but quite expensive otherwise. This program illustrates the respective costs of `+` and `StringBuffer.append()`:

```
// string append
public class str_app {
        public static void main(String args[]) {
                final int N = 10000;

                // using +

                Timer t = new Timer();
                String s1 = "";
                for (int i = 1; i <= N; i++)
                        s1 = s1 + "*";
                t.print("append using +");

                // using StringBuffer

                t.reset();
                StringBuffer sb = new StringBuffer();
                for (int i = 1; i <= N; i++)
                        sb.append("*");
                String s2 = sb.toString();
                t.print("append using StringBuffer");
        }
}
```

This program takes around 2400 units of time to run using the + operator, and about 60 units of time using `StringBuffer.append()`, a difference of 40-1.

4.2 Accumulating Strings Using char[] Arrays

String buffers are an effective way of accumulating strings, that is, building up a string a piece at a time. Another approach to achieve the same end, that is useful if you have a rough idea of the maximum size of the output string, is direct use of `char[]` arrays. In other words, you create an output `char[]` array, add characters to it, and then convert the result to a string.

An application that uses this technique is one where you convert from the UTF-8[3] encoding used by Java libraries (when storing Java strings to a disk file) to a string object. Java characters are encoded as one, two, or three bytes in the UTF-8 scheme, so when a stream of bytes is converted back to characters, it is guaranteed that the number of characters will never be more than the number of bytes. The following code takes advantage of this idea:

```
// convert UTF-8 to String

static String UTFtoString(byte b[]) {
        int maxlen = b.length;
        char str[] = new char[maxlen];
        int outlen = 0;
        for (int i = 0; i < maxlen; i++) {
                byte c = b[i];
                if ((c & 0x80) == 0) {
                        str[outlen++] = (char)c;
                }
                else if ((c & 0xf0) == 0xe0) {
                        str[outlen++] = (char)(((c & 0xf) << 12) |
                            ((b[i+1] & 0x3f) << 6) |
                            (b[i+2] & 0x3f));
                        i += 2;
                }
                else {
                        str[outlen++] = (char)(((c & 0x1f) << 6) |
                            (b[i+1] & 0x3f));
                        i++;
                }
        }
        return new String(str, 0, outlen);
}
```

A `char[]` array is created with length equal to the number of input bytes, and then characters are accumulated into the array, with any extra positions in the array simply left unused, and the array converted to a string at the end of processing. This particular technique trades some extra memory usage for improved speed. Another approach would be to go through the byte array and look at the bit patterns, such that the number of characters implied by the byte sequences could be totalled, and then a `char[]` array of exactly the right length allocated.

[3] See 22.1.15 in the *Java™ Language Specification*.

4.3 Using == and String.equals() to Compare Strings

If you've programmed in languages such as C++, that support overloaded operators, you might be used to using the `==` operator to compare strings. You can also use this operator in Java programming, but it won't necessarily give you the results you expect. In the Java language, the `==` operator, when applied to references, simply compares the references themselves for equality, and *not* the referenced objects. For example, if you have strings:

```
String s1 = "abc";
String s2 = "def";
```

then the boolean expression:

```
s1 == s2
```

will be false, not because the string contents are unequal, but because the `s1` and `s2` references are so. Conversely, if two references are equal using `==`, then you can be sure that they refer to identical objects. So if you are comparing strings, and there is a good chance of encountering identical ones, then you can say:

```
if (s1 == s2 || s1.equals(s2))
   ...
```

If the references are identical, this will short-circuit the `equals()` method call.

This technique illustrates a more general principle of performance – always perform a cheap test before an expensive one, if you possibly can. In this example, `==` is much less expensive to perform than `equals()`.

4.4 Interning Strings

The idea in the previous section can be taken further, with the notion of *interning*. The Java `String` class has a method called `intern()`, used to create an internal pool of unique strings. Given a string `str`, saying:

```
str = str.intern();
```

adds the string to the internal pool if not already there, and returns a reference to the interned string. Interned strings can be compared with each other using `==`, which is much cheaper than using `equals()` to do the comparison. String literals are always interned.

An example of interning strings:

```
// interning strings

public class str_intern {
        public static void main(String args[]) {
                String str = new String("testing");

                // interning not done

                if (str == "testing")
                        System.out.println("equal");
                else
                        System.out.println("unequal");

                // interning done

                str = str.intern();
                if (str == "testing")
                        System.out.println("equal");
                else
                        System.out.println("unequal");
        }
}
```

In the first example, the test `str == "testing"` compares false, because `str` is a reference to a string that is a *copy* of the literal "testing", while "testing" itself is an interned string, and therefore the corresponding references are distinct. In the second example, the test compares true, because `str` has been interned, and the reference returned from the `intern()` call is identical to the reference to the "testing" literal already in the internal string pool.

String interning is a very fast way of doing comparisons within a pool of strings, with an initial cost incurred in actually doing the interning.

4.5 Obtaining the Length of a String

`length()` is a method in `String` used to obtain the number of characters currently in the string. You might not think that there is any performance angle with this simple method, but it turns out that if you call `length()` in a tight loop, there can be a performance hit. The same idea applies to any method called on a per-character basis, such as `charAt()`.

Here is an example that illustrates this idea:

```
// string length()

public class str_len {
        public static void main(String args[]) {
                final int N = 1000000;

                StringBuffer sb = new StringBuffer();
                for (int i = 1; i <= N; i++)
                        sb.append('*');
                String s = sb.toString();

                // length()

                int cnt = 0;
                Timer t = new Timer();
                for (int i = 0; i < s.length(); i++) {
                        if (s.charAt(i) == 'x')
                                cnt++;
                }
                t.print("using length()");

                // precomputed length

                cnt = 0;
                t.reset();
                for (int i = 0, len = s.length(); i < len; i++) {
                        if (s.charAt(i) == 'x')
                                cnt++;
                }
                t.print("using precomputed len");
        }
}
```

The first example using `length()` takes 1733 units of time, while if the string length is precomputed and stored in a local variable, the total time is 1282 units. It's more costly to call a method than it is to precompute the length. `length()` is a likely candidate for inline optimization, so the results you see with your compiler may not match those reported here.

4.6 Using toCharArray()

`charAt()` is a `String` method similar to `length()`, which can be expensive if called for every character in a long string. An alternative to this method is use of `toCharArray()`, which returns a copy of the string in a `char[]` array.

An example that uses this method looks like this:

```
// string charAt() and toCharArray()

public class str_tochar {
        public static void main(String args[]) {
                final int N = 1000000;

                StringBuffer sb = new StringBuffer();
                for (int i = 1; i <= N; i++)
                        sb.append('*');
                String s = sb.toString();

                // using charAt()

                int cnt = 0;
                Timer t = new Timer();
                for (int i = 0, len = s.length(); i < len; i++) {
                        if (s.charAt(i) == 'x')
                                cnt++;
                }
                t.print("using charAt()");

                // using toCharArray()

                t.reset();
                cnt = 0;
                char ss[] = s.toCharArray();
                for (int i = 0; i < ss.length; i++) {
                        if (ss[i] == 'x')
                                cnt++;
                }
                t.print("using toCharArray");
        }
}
```

The first part of the example takes 1282 units of time to complete, while the second part takes 481. In other words, exporting the characters in a string to a `char[]` array eliminates the method call overhead of `charAt()`, at the expense of committing extra memory to the `char[]` copy that is made.

Note also in this example that calling a method `length()` is not the same as obtaining the length of an array via the special `length` field. The former has the expense of method call overhead, whereas the latter simply retrieves the array length from a run time descriptor. The Java Virtual Machine has a bytecode `arraylength` that is used for this purpose.

4.7 Converting Strings to Numbers

Suppose you have a string like "12.34" that represents a number, and you'd like to convert it to numeric form. How expensive is such an operation? One way to find out is by writing a small program that creates a `Double` (a wrapper class for `double`), from either a string or a number.

The program looks like this:

```
// converting strings to numbers

public class str_double {
        public static void main(String args[]) {
                final int N = 100000;
                Double d;

                // construct Double from string

                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        d = new Double("12.34");
                t.print("as string");

                // construct Double from number

                t.reset();
                for (int i = 1; i <= N; i++)
                        d = new Double(12.34);
                t.print("as number");
        }
}
```

Creating a `Double` from a string takes about 15 times as long as from a number. This difference starts to make sense when you consider all the processing that must go on when converting to a number, including operations such as retrieving characters from the string, multiplication, handling the decimal point, and so on.

Often you have no choice but to do conversion of strings to numbers, but it's worth keeping in mind the expense of this operation.

5 Input and Output

I/O is a fundamental aspect of many programs, and there are several important techniques for speeding up input and output in Java applications.

5.1 Buffering

Perhaps the most important idea in improving I/O performance is *buffering*, doing input and output in large chunks instead of a byte or character at a time. To see what difference this can make, consider the following program:

```
// buffered I/O

import java.io.*;

public class io_buf {
        public static void main(String args[]) {
                try {
                        // one read() per character

                        FileInputStream fis =
                            new FileInputStream(args[0]);
                        int cnt = 0;
                        int c;
                        Timer t = new Timer();
                        while ((c = fis.read()) != -1)
                                if (c == 'x')
                                        cnt++;
                        t.print("read() per character");
                        fis.close();

                        // buffered

                        fis = new FileInputStream(args[0]);
                        byte buf[] = new byte[1024];
                        cnt = 0;
                        int n;
                        t.reset();
                        while ((n = fis.read(buf)) > 0)
                                for (int i = 0; i < n; i++)
                                        if (buf[i] == 'x')
                                                cnt++;
                        t.print("buffered");
                        fis.close();
                }
                catch (IOException e) {
                        System.err.println(e);
                }
        }
}
```

This program uses two different approaches to count the number of "x" bytes in a file. The first repeatedly calls `read()` on the input stream to grab individual bytes, while the second reads 1024-byte chunks of the file and then iterates through each chunk, and counts the bytes that way.

The second of these approaches is about 20 times faster than the first, because the buffering approach avoids method call overhead, and because each call to `read()`, whether for a single byte or a large chunk of bytes, results in a call to the underlying operating system (the `read()` methods are declared `native` in `java.io.FileInputStream`). Method calls are relatively expensive, as are calls into the underlying operating system.

5.2 BufferedReader

We can build on the idea in the previous section. I/O classes found in `java.io` are organized in a layered fashion, that is, one I/O layer can be added on top of another, to provide additional functionality. This idea is quite important when applied to performing input and output efficiently. To see why this is, consider an example:

```
// BufferedReader

import java.io.*;

public class io_buf2 {
        public static void main(String args[]) {
                try {
                        // without buffering

                        Reader r =
                            new FileReader(args[0]);
                        Timer t = new Timer();
                        int c;
                        while ((c = r.read()) != -1)
                                ;
                        t.print("without buffering");
                        r.close();

                        // with buffering

                        r = new FileReader(args[0]);
                        r = new BufferedReader(r);
                        t.reset();
                        while ((c = r.read()) != -1)
                                ;
                        t.print("with buffering");
                        r.close();
                }
                catch (IOException e) {
                        System.err.println(e);
                }
        }
}
```

The example has two parts. The first reads all the characters from a file using a `FileReader` class, and the second also reads all characters, but this time with a `BufferedReader` layered on top of the `FileReader`.

The second approach runs about four times as fast as the first, because there's less overhead incurred per character. `BufferedReader` reads characters in chunks several thousand long, and then parcels them out one at a time. In other words, the `read()` method found in `BufferedReader` simply hands back characters from a buffer, which is quite efficient.

5.3 Formatting

We often tend to focus on the actual costs of data input or output, while neglecting the associated cost of formatting that data. In other words, formatting costs relate to arranging characters into specific meaningful patterns, which are then sent to a disk file or across a network.

One way of exploring the costs of formatting is by writing a series of examples, each of which results in identically-formatted text, but with different approaches used for formatting. In each case, we want to format lines of the type:

```
The square of 5 is 25
The square of 6 is 36
...
```

The first approach uses string concatenation:

```
// format #1

public class io_msg1 {
        public static void main(String args[]) {
                final int N = 25000;
                String s;
                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        s = "The square of " + i + " is " + (i * i);
                t.print("format using +");
        }
}
```

The second approach uses `java.text.MessageFormat`, with a precompiled format string:

```
// format #2

import java.text.*;

public class io_msg2 {
        public static void main(String args[]) {
                final int N = 25000;
                Object argvec[] = new Object[2];
                MessageFormat f = new MessageFormat(
                    "The square of {0,number,#} is {1,number,#}");
                Timer t = new Timer();
                for (int i = 1; i <= N; i++) {
                        argvec[0] = new Integer(i);
                        argvec[1] = new Integer(i * i);
                        String s = f.format(argvec);
                }
                t.print("format using precompiled");
        }
}
```

The third approach is similar to the second, but uses an on-the-fly message format that has not been precompiled:

```
// format #3

import java.text.*;

public class io_msg3 {
        public static void main(String args[]) {
                final int N = 25000;
                Object argvec[] = new Object[2];
                String f =
                    "The square of {0,number,#} is {1,number,#}";
                Timer t = new Timer();
                for (int i = 1; i <= N; i++) {
                        argvec[0] = new Integer(i);
                        argvec[1] = new Integer(i * i);
                        String s = MessageFormat.format(f, argvec);
                }
                t.print("format using non-precompiled");
        }
}
```

How do these compare in formatting costs? The times are as follows:

```
#1       1953

#2       12027

#3       41409
```

or about 20-1 between the slowest and fastest. This doesn't prove that you should avoid using `MessageFormat`; it's quite an important tool if you're trying to internationalize your applications. But it's worth understanding the relative costs of different formatting approaches.

5.4 Obtaining File Information

Like other programming languages, the Java libraries rely on support from the underlying operating system. This consideration is important when you're working with data files in a Java application. For example, consider the following simple program:

```
// display the length of a file

import java.io.*;

public class io_length {
        public static void main(String args[]) {
                long len = new File(args[0]).length();
                System.out.println(len);
        }
}
```

The program calls `File.length()` to obtain the length of a file that is specified on the command line. `length()` itself doesn't know the length, and so it calls upon the underlying operating system to retrieve the length, which might, for example, be obtained via a `stat()` system call. Such calls are relatively

expensive, and it's worth keeping this in mind when you're querying information about files (such as file length, whether a file is a directory or a plain file, attributes such as read/write permissions, and so on).

One simple expedient is to avoid querying the same information twice. For example, if you've already obtained a file's length, don't call for it again a few lines down in your program, unless you have reason to believe the length has changed in the meantime.

6 Libraries

This section touches on some of the performance issues with using classes and methods from the standard Java libraries.

6.1 System.arraycopy()

`System.arraycopy()` is a method that supports efficient copying from one array to another. For example, if you have two arrays `vec1` and `vec2`, of length `N`, and you want to copy from `vec1` to `vec2`, you say:

```
System.arraycopy(vec1, 0, vec2, 0, N);
```

specifying the starting offset in each array.

It's worth asking how much `System.arraycopy()` improves performance, over alternative approaches for copying arrays. Here is a program that uses a copy loop, `System.arraycopy()`, and `Object.clone()` to copy one array to another:

```
// copying arrays

public class lib_copy {
        public static void main(String args[]) {
                final int N = 1000000;
                int vec1[] = new int[N];
                for (int i = 0; i < N; i++)
                        vec1[i] = i;
                int vec2[] = new int[N];

                // copy using loop

                Timer t = new Timer();
                for (int i = 0; i < N; i++)
                        vec2[i] = vec1[i];
                t.print("loop");

                // copy using System.arraycopy()

                t.reset();
                System.arraycopy(vec1, 0, vec2, 0, N);
                t.print("System.arraycopy");

                // copy using Object.clone()

                t.reset();
                vec2 = (int[])vec1.clone();
                t.print("clone");
        }
}
```

The times for the various methods are:

```
loop                    381

System.arraycopy        40

Object.clone            150
```

If you use a just-in-time compiler for this example, the difference between the loop and `System.arraycopy()` shrinks substantially, but `System.arraycopy()` is still faster.

For very short arrays, use of this method may be counterproductive, because of overhead in actually calling the method, checking the method's parameters, and so on.

`Object.clone()` represents another approach to copying an array. `clone()` allocates a new instance of the array and copies all the array elements. Note that `Object.clone()` does a *shallow* copy, as does `System.arraycopy()`, so if the array elements are object references, the references are copied, and no copy is made of the referenced objects.

6.2 Vector vs. ArrayList

The class `java.util.Vector` is used to represent lists of object references, with support for dynamic expansion of the vector, random access to vector elements, and so on.

A newer scheme is the Java collection framework, which includes a class `ArrayList` that can be used in place of `Vector`. Some of the performance differences between `Vector` and `ArrayList` include:

- `Vector`'s methods are synchronized, `ArrayList`'s are not. This means that `Vector` is thread-safe, at some extra cost (§3.3).

- The collection framework provides an alternative to `ArrayList` called `LinkedList`, which offers different performance tradeoffs (§6.4).

- When `Vector` needs to grow its internal data structure to hold more elements, the size of the structure is doubled, whereas for `ArrayList`, the size is increased by 50%. So `ArrayList` is more conservative in its use of space.

It's worth using the collection framework in your applications if you possibly can, because it's now the "standard" way to handle collections. If you are concerned about thread safety, one way to handle this issue is to use wrappers around objects like `ArrayList`, for example:

```
List list = Collections.synchronizedList(new ArrayList());
```

This technique makes `list` thread-safe.

6.3 Setting Initial Array Capacity

Collection classes like `ArrayList` periodically must grow their internal data structures to accommodate new elements. This process is automatic, and normally you don't need to worry about it. But if you have a very large array, and you know in advance that it's going to be large, then you can speed things up a bit by calling `ensureCapacity()` to set the size of the array. An example:

```
// ensureCapacity()

import java.util.*;

public class lib_cap {
        public static void main(String args[]) {
                final int N = 1000000;
                Object obj = new Object();

                ArrayList list = new ArrayList();
                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        list.add(obj);
                t.print("without ensurecapacity");

                list = new ArrayList();
                t.reset();
                list.ensureCapacity(N);
                for (int i = 1; i <= N; i++)
                        list.add(obj);
                t.print("with ensurecapacity");
        }
}
```

Calling `ensureCapacity()` means that `ArrayList` will not have to keep growing the internal structures as list elements are added. The second case above runs about 15% faster than the first. Of course, if you call `ensureCapacity()` when you don't really need it, you may end up wasting a lot of space.

6.4 ArrayList vs. LinkedList

The Java collection framework provides two classes for handling lists of data items, `ArrayList` and `LinkedList`. The first of these is conceptually like an array, the second like a linked data structure. An `ArrayList` is implemented using an internal array of `Object[]`, while a `LinkedList` uses a series of internal records linked together. These two classes have very different performance characteristics, as illustrated by a couple of examples.

The first deals with inserting new elements at position 0 in a list:

```
// ArrayList vs. LinkedList #1

import java.util.*;

public class lib_list1 {
        public static void main(String args[]) {
                final int N = 25000;

                // ArrayList

                ArrayList al = new ArrayList();
                Timer t = new Timer();
                for (int i = 1; i <= N; i++)
                        al.add(0, new Integer(i));
                t.print("arraylist");

                // LinkedList

                LinkedList ll = new LinkedList();
                t.reset();
                for (int i = 1; i <= N; i++)
                        ll.add(0, new Integer(i));
                t.print("linkedlist");
        }
}
```

In this first example, the times are as follows:

```
ArrayList      6610

LinkedList     340
```

Inserting elements at the beginning of an `ArrayList` requires that all existing elements be pushed down. This is a very expensive operation.

But inserting at the beginning of `LinkedList` is cheap, because the elements of the structure are connected with each other via links, and it's easy to create a new element and link it in with the current element at the head of the list.

The second example does random lookup of elements already in a structure:

```
// ArrayList vs. LinkedList #2

import java.util.*;

public class lib_list2 {
        public static void main(String args[]) {
                final int N = 25000;
                Object o;

                // ArrayList

                ArrayList al = new ArrayList();
                for (int i = 0; i < N; i++)
                        al.add(new Integer(i));
                Timer t = new Timer();
                for (int i = 0; i < N; i++)
                        o = al.get(i);
                t.print("arraylist");

                // LinkedList

                LinkedList ll = new LinkedList();
                for (int i = 0; i < N; i++)
                        ll.add(new Integer(i));
                t.reset();
                for (int i = 0; i < N; i++)
                        o = ll.get(i);
                t.print("linkedlist");
        }
}
```

The running times here are:

```
ArrayList      30

LinkedList     42371
```

In this example, there's a premium on being able to randomly access the elements of a list, something that's cheap to do with an array, but expensive with a linked list (because you must iterate over the elements of the list to find the right one).

So we can say that `ArrayList` offers superior performance if you're mostly concerned with adding elements to the end of the list, then looking them up randomly, whereas `LinkedList` is a better choice if you're adding and deleting elements at the front or middle of the list, and accessing elements mostly in sequential order via iterators that step through the list elements.

6.5 Programming in Terms of Interfaces

The example in the previous section illustrates an important point, namely, that there may be more than one "right" way to represent data. You might, for example, decide that an `ArrayList` is the best choice for some application, but then later decide that a `LinkedList` would in fact be a better choice.

One way you can make changeovers like this easier to program is by using interface types, instead of actual types of classes that implement the interface. For example, if you say:

```
List list = new ArrayList();
```

instead of saying:

```
ArrayList list = new ArrayList();
```

and you do likewise when declaring method parameters, as in:

```
void f(List list) {...}
```

instead of:

```
void f(ArrayList list) {...}
```

then you can use `list` everywhere in your programming, and you don't have to worry too much about whether `list` is really an `ArrayList` or a `LinkedList` (it might actually be a custom-designed collection class, in fact). In other words, you program in terms of functionality offered by interface types, and you can switch the actual implementation easily at some later point, for reasons of efficiency.

7 Size

In this section we look at a variety of techniques for reducing the disk and memory usage of Java programs.

7.1 .class File Size

Java compilers generate `.class` files, with each file containing bytecodes for the methods in one class. These files contain a series of sections, or attributes, and it's possible to exercise some control over which sections actually show up in a `.class` file. The `Code` attribute contains actual bytecodes for methods, `SourceFile` information on the source file name used to generate this `.class` file, `LineNumberTable` mapping information between bytecode offsets and source file line numbers, and `LocalVariableTable` mapping information between stack frame offsets and local variable names. You can use options to the `javac` compiler to control which sections are included:

```
javac          Code, SourceFile, LineNumberTable

javac –g       Code, SourceFile, LineNumberTable, LocalVariableTable

javac -g:none  Code
```

How much do these options affect code size? A test using the JDK 1.2.2 sources in the `java.io`, `java.lang`, and `java.util` packages was run, with total sizes of all `.class` files as follows:

```
javac          668K

javac –g       815K

javac –g:none  550K
```

So use of `javac –g:none` saves about 20% in space over the default `javac`, and using `javac –g` is about 20% larger than the default. There are also commercial tools that tackle this particular problem.

If you strip out information such as line numbers from `.class` files, you may have problems with debugging and exception reporting.

7.2 Wrappers

The Java libraries have a variety of what are known as *wrapper classes*, classes used to hold a value of a primitive type such as `int` or `double`. For example, you can say:

```
Integer obj = new Integer(37);
```

to create a wrapper class instance. This instance can then be assigned to an `Object` reference, have its type tested using `instanceof`, used with collection classes like `ArrayList`, and so on.

One drawback, however, is that wrapper classes have some overhead, both in time and space costs. It's less efficient to extract a value from a wrapper instance than it is to use the value directly. And there are costs in creating all those wrapper instances, both time costs in calling `new` and constructors, and space costs with the overhead that comes with class instances (a very rough estimate might be that an instance of a class takes 15 bytes, in addition to the size of the actual instance variables).

7.3 Garbage Collection

The Java language uses a form of automatic memory management known as *garbage collection*. Memory for objects and arrays is allocated using `new`, and then reclaimed when no longer in use. Garbage collection is a process of detecting when an object no longer has any references to it, with the object's memory then reclaimed and made available as free space.

Normally, you don't need to worry about garbage collection, but there are cases where it's possible to help the process along a bit. For example, suppose that you have a stack of object references, along with a current stack index, and you pop an entry off the stack:

```
Object pop() {
   return stack[stackp--];
}
```

This works pretty well, but what happens to the slot in the stack that was just popped? It still has a reference to some object type, and the object may be very large (like a big array). An identical reference has been passed back to the caller of `pop()`, but that caller may quickly process and discard the reference, and thus it's possible that the object reference could be garbage collected, except for one thing – there's still a reference to the object on the stack. So a better way to write this code is:

```
Object pop() {
   Object tmp = stack[stackp];
   stack[stackp--] = null;
   return tmp;
}
```

In other words, the reference has been set to `null`. If this was the last reference to the object, then the object is made available for reclaiming by the garbage collector.

This idea can be generalized. If you have a reference to a large, no-longer-used object, and the reference is one that will stay in scope for some time to come, then you might want to clear the reference explicitly, by setting it to `null`. For example, you might have a class instance that persists throughout the life of your application, and one of the instance's variables references a large object, an object you are no longer using.

7.4 SoftReference

`java.lang.ref.SoftReference` is a relatively new class, used to implement *smart caches*. A smart cache is storage of some type of data, such as a file that's been read from disk, which is released by the Java run time system if memory is needed for other purposes. In other words, suppose that there's a need for additional memory in a running Java application. The garbage collection system in the Java Virtual Machine knows about soft references, and it will clear the object reference referred to by a soft reference, before throwing an `OutOfMemoryError` exception.

To see how this works, suppose that you have an `Object` reference that points to a large array:

```
Object obj = new char[1000000];
```

and you'd like to keep this array around if possible, but you're willing to let it go if memory is desperately short. You can use a soft reference:

```
SoftReference ref = new SoftReference(obj);
```

`obj` is the *referent* of the soft reference.

Then, at some later point, you check the reference by saying:

```
if (ref.get() == null)
   (referent has been cleared)
else
   (referent has not been cleared)
```

If the referent has been cleared, then the garbage collector has reclaimed the space used by it, and your cached object is gone. Note that if the referent has other references to it, the garbage collector will not clear it. In other words, if the only reference to an object is via a `SoftReference` object, and the object's memory is needed elsewhere in the program, then the object is subject to being cleared by the garbage collector.

This scheme can be used to implement various types of caches, where objects are kept around as long as possible, but cleared if memory is tight.

7.5 BitSet

`BitSet` is a class used to represent arrays of bits efficiently. A simple example of its use is:

```
import java.util.*;

public class size_bitset {
        public static void main(String args[]) {
                BitSet b = new BitSet();
                b.set(37);
                b.set(59);
                b.set(73);
                System.out.println(b);
        }
}
```

with output of:

```
{37, 59, 73}
```

`BitSet` differs from a primitive array of `boolean` in two ways: (1) a `BitSet` object is dynamic, that is, the set of bits can be added to at any time, and (2) a `BitSet` is at least potentially more space efficient, in that an array of `boolean` may in fact be implemented as an array of `byte`, using 8 bits per value instead of 1[4].

7.6 Sparse Arrays

Suppose you have a large array that you'd like to store, but it's so large that you are worried about obtaining a contiguous block of space for it, or perhaps it's large but sparse (few actual elements are set), and you'd like a more efficient way of storing the array.

One way of doing this is illustrated by the following example:

[4] JDK 1.2.2 stores `boolean` arrays as `byte` arrays.

```
public class SparseArrayList extends java.util.AbstractList {
        private static final int PAGE_SIZE = 1024; // page size
        private Object pages[][] = new Object[0][]; // pages

        // constructor
        public SparseArrayList() {}

        // return total size of allocated pages
        public int size() {
                return pages.length * PAGE_SIZE;
        }

        // set an array slot to a given value
        public Object set(int index, Object val) {
                if (index < 0)
                        throw new IllegalArgumentException();
                int p = index / PAGE_SIZE;
                if (p >= pages.length) { // expand pages list
                        Object newpages[][] = new Object[p + 1][];
                        System.arraycopy(pages, 0, newpages, 0,
                            pages.length);
                        pages = newpages;
                }
                if (pages[p] == null) // allocate new page
                        pages[p] = new Object[PAGE_SIZE];
                Object old = pages[p][index % PAGE_SIZE];
                pages[p][index % PAGE_SIZE] = val;
                return old;
        }

        // get the value of a given array slot, null if none
        public Object get(int index) {
                if (index < 0)
                        throw new IllegalArgumentException();
                int p = index / PAGE_SIZE;
                if (p >= pages.length || pages[p] == null)
                        return null;
                return pages[p][index % PAGE_SIZE];
        }
}
```

The class `SparseArrayList` extends the collection framework abstract class `AbstractList`, and thereby provides a standard `List` interface to the user.

The array is chopped up into pages, each page with 1024 elements. An internal table is kept of the pages, with a page allocated only as needed. A passed-in array index is divided into page number and page offset, and a given array element is stored based on the number and offset. For example, an index of 1030 specifies page 1, offset 6.

An example of using `SparseArrayList` looks like this:

```
import java.util.Random;

public class size_driver {
        public static void main(String args[]) {
                SparseArrayList list = new SparseArrayList();
                Random rn = new Random();
                for (int i = 1; i <= 1000000; i++) {
                        int r = rn.nextInt(1000000);
                        list.set(r, new Integer(r));
                        Integer iw = (Integer)list.get(r);
                        if (iw.intValue() != r)
                                System.err.println("error");
                }
        }
}
```

This is a test program for `SparseArrayList`; it sets array slots at random, then checks whether the slot that has been set has the appropriate value in it.

There are other areas where you might want to design your own class that extends the collection framework. For example, you might be trying to represent data that contains long sequences of identical elements (run length encoding), and you can devise a data structure that handles this type of data efficiently.