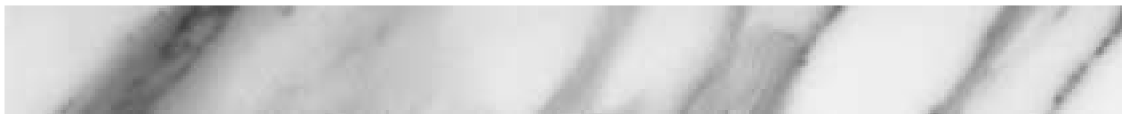
PERFORMANCE IN GENERAL

Topics in This Chapter

- When to consider performance
- Categorizing performance problems

Chapter 1



An ounce of prevention is worth a pound of cure.

Anonymous proverb

This chapter considers the relationship of performance considerations to the coding process and introduces a way of categorizing performance problems that will be used throughout this book. The first question that may come to mind is, "Just what do you mean by performance?"

There are really two ways to view the performance of a program. There is the objective view: "The program took 3 seconds to run and occupied 8 megabytes of memory." In the objective view, we can measure performance as the time it takes for a particular program to execute upon a particular piece of hardware. Other objective measurements of performance might include the resources a program consumes in its execution such as memory size. The objective measurements can apply to both the speed of the program and to the total cost of running the program.

Then there is the subjective view: "The program was fast enough for our needs." In the subjective view, we define performance relative to our needs. A program may perform to the satisfaction of one user but not to the satisfaction of another. How well a program performs will always be relative to the expectations of the program's users.

1.1 Performance Versus Optimization

You will often see advice stating that you really shouldn't worry about performance until you have the code working. These statements are generally referring to the fact that you should not worry about the low-level optimization of code. Spending a lot of time optimizing any given code path is usually a waste of time. It is often unclear how much any single piece of code will be executed. The larger the program, the more this is true.

However, this does not mean that you shouldn't think about the overall performance requirements of your program from the very start. If you have no idea what the performance requirements for your program are, then you will most likely produce a program that will not meet the goal. For example, if your program needs to monitor an assembly line once a second, this is important information that will affect the design of the code.

As with most endeavors, a balance must be struck. It is quite likely that if you wait until after the design and code are done to think about performance, then you will end up with code that has major problems. On the other hand, if you spend all your time optimizing every last line of code, you will never ship a product.

A good clean design is very important. Simple, readable code is also very important. Luckily, in many cases, there is no conflict between code simplicity and performance. The simplest code usually performs best. In many cases, it is merely a matter of learning to write code in a style that performs well. In Chapter 3, we will give some simple examples of Java performance characteristics. In the rest of the book, we will build up your repertoire of performance styles.

If it is unclear just how often a particular piece of code is going to be used, then it is probably best not to spend time thinking of clever ways to ensure that it has the best possible performance. In this case, write something reasonable and revisit it for performance later if you need to. Note, however, that we do try to stress the word *reasonable* here. If you write code that is hopeless, it will almost always come back and haunt you.

In some cases, it will be quite clear at design time that a piece of code will be used very often. If this is the case, then it is worth spending some time thinking about how this piece of code might perform. Think about not only the details of the code you are writing but also how it interacts with the rest of the program. In this case, you would be optimizing your program before the fact. This should be done cautiously and only when the performance implications are crystal clear.

1.2 Performance Life Cycle

There are many ways to approach software design. Many fine books have been written about software design, so we will not talk about any specific design methodology. What we have noticed, however, is that many of the approaches to software design can be summarized into three basic steps: steps you take before coding, steps you take during coding, and steps you take after you have coded. Then in the development process, you iterate back and forth among these three steps.

Just as software design breaks down reasonably well into these three basic steps, the steps involved in getting your software to perform also break down into these same three steps.

Before Coding

The easiest performance problems to fix are the ones that aren't there in the first place. If the performance of your code is a goal you set for yourself from the beginning of your project, then it is likely that you will be well on the way to having the performance you desire.

Before you start any coding is the time to understand the basic goals of the software you are going to produce. This is obviously true from a functional point of view—you need to have some understanding of what you are going to write before you write it. It is equally true from a performance point of view.

The performance needed by a program is defined by the nature of the program. Some performance requirements are explicit, such as:

- The system must handle 1000 order requests per hour.
- The temperature of the engine must be checked every 3 seconds.

Other requirements are not so crisp:

- The user should experience a smooth flow of control.
- The background process should not noticeably interfere with interactive jobs.

When you are given requirements that are not crisp, try to discover what specific requirements are really intended. After you have these parameters set, you will have come a long way toward determining how much effort to

invest. For example, on further investigation, you might find that the requirement

- The user should experience a smooth flow of control.

actually means that

- The user should not experience a delay of more than 0.25 seconds when navigating between screens.

In addition to simple statements of time, make sure that there is a requirement stating on what kind of hardware the user expects to achieve a given level performance. There are worlds of difference between a typical desktop machine and a high-end server system. There are indeed real reasons why a server box is more expensive than a simple desktop machine. Find out as much as you can about the expected hardware that will run your application, including processor speed, memory size, disk configurations, network speed, and anything else that may have a bearing on your code.

It is important that you determine the basic performance requirements of time and platform during the design phase of your project. This will be very helpful as a reality check for the rest of your design. If you know that your code must be constrained to run with a particular amount of memory, then that will limit some of the data types or algorithms you use. We'll talk more about some common Java data types in Chapter 3.

Simple Design Guidelines

In addition to hard performance requirements like time and resources, it is useful to keep a few guidelines in mind as you approach the design of your code.

- *Always use the simplest classes possible—but no simpler.* Designing code is a perpetual exercise of balance. Picking a data structure that is too complex for the task at hand can lead to poorly running and scaling code. For example, picking a *java.util.Vector* when an array would have met all of your needs is not a good decision. In Chapter 3, we will show some examples of the performance differences these simple choices can make. However, using a data structure that does not meet the needs of the design can be just as bad. An inadequate data structure will lead to code duplication and the loss of many of

the advantages of working in an OO language in the first place. If your code really needs the functionality of a *java.util.Vector*, then using an array will likely just lead you to recode a *java.util.Vector* within your own class.

- *Never do for yourself what you can get someone else to do for you.* If someone else has already written classes that do what you want, then make use of them. Just remember that *what you want* includes meeting your performance characteristics. A good example at a lower level of detail often comes in exception handling. Your caller may be able to handle a particular exception better than you can. If the only thing you can do after catching an exception is to rethrow it without adding any value, then you have simply wasted time catching the exception.
- *Never do today what can be put off until tomorrow.* If a design or piece of code is proving to be exceptionally difficult to write, then it may be a good point to delegate the hard parts to another class. A good example of how this rule of thumb can save in performance costs can be found in the translation of locale-specific information. For example, when an error message that has human readable information in it is sent, there is seldom any purpose in translating it into a specific language until the message is actually displayed to a human. So, it is often best to leave the final message translation to the client code, not at the time an exception (for example) is first thrown.

Design Notes

As you proceed with the design of your project, make sure to put any knowledge you have about performance needs into the design. This can be a great aid during coding as you try to anticipate which sections may impact performance the most. If, for example, you know at design time that a particular part of the application gets used more than others, then make a note of this. Even if the designer and the coder are the same person, these pieces of information can be quite helpful. If the designer and coder are not the same person, the information can be critical.

During Coding

A complaint we hear quite often is: “We just want to get the code to function and then we’ll fix the performance. So go away until I’ve gotten it written.”

To one extent, we agree with this in that if the code does not function correctly then it does not matter much how quickly it works. However, the statement really just exposes a misconception about the nature of well-performing code. The misconception is that well-performing code is by nature hard to write. On the contrary, well-performing code is often not any harder to write than poorly performing code. In fact, it is often easier. The trick is to learn and use coding habits that contribute to well-performing code.

In Chapter 3, we show some of the more common simple Java coding techniques for both writing code that will perform well and avoiding code that will not perform well. If you practice these coding tips, you will find that it takes no extra time to follow them. Some areas to pay attention to follow.

- *Code that is in a loop.* If the code is in a loop, then it will be executed many times. Don't recalculate things that are going to be constant for the life of the loop. If there are ways to break out of the loop early, take them.
- *Object creation.* Creating an object ties up resources in both time and space. Make sure that you really need to create the object. Often you can reuse an existing object rather than create a new instance each time.
- *Code-using collections.* Choosing the right kind of collection to hold your objects can have a major impact on the performance of your application.

Code Choices

As you are coding, you may realize that there are various choices on how to code some particular function. In the following class *Primes,* we show a couple of ways to implement the Sieve of Eratosthenes. This is a simple technique for finding all the primes less than or equal to a given number. It is called a sieve because it sifts out the prime numbers by removing nonprimes.

In method *eratosthenes1,* we use a *java.util.BitSet* in which if the i-th bit is true, then the number $(i \times 2) + i$ is prime. Since the constructor of a *java.util.BitSet* gives us a *java.util.BitSet* with all the bits set to false, we must first set all the bits to true. To the user of *eratosthenes1*, all the details of how the primes are computed are hidden. In method *eratosthenes2*, we reverse this logic and decide that a value of false in the *java.util.BitSet* indicates that the number is prime. Since the constructor gives us false values to start with, we can use the constructed *java.util.BitSet* with no further initialization. This is a nice example of using the guideline to never do for yourself

what someone else will do for you. In this case, Java has already conveniently initialized the *java.util.BitSet*—you don't need to do it again.

In the method *eratosthenes3*, we change the *java.util.BitSet* to a boolean array. Functionally, this serves just as well as the *java.util.BitSet*, in this case. Remember always to use the simplest class possible.

We can run *Primes* by typing:

```
java Primes
```

Doing this on Machine A[1] produces the following output:

```
eratosthenes1 found 78498 primes in the first 1000000 numbers in 1212 milliseconds
eratosthenes2 found 78498 primes in the first 1000000 numbers in 801 milliseconds
eratosthenes3 found 78498 primes in the first 1000000 numbers in 151 milliseconds
```

By making the simple change to the code from *eratosthenes1* to *eratosthenes2*, we reduce the time spent by about 30%. Since all the details of the implementation are hidden from the user, we are completely free to make changes like this. Changing the the *java.util.BitSet* to a boolean array from *eratosthenes2* to *eratosthenes3* reduces the time spent by an additional 81%.

These changes were both quite simple and produced quite dramatic results. This is not an uncommon occurrence. It is often the case that moving to simpler structures and taking advantage of already set values will produce excellent results.

The class *Primes* containing the methods *eratosthenes1*, *eratosthenes2*, and *eratosthenes3* follows.

```
import java.util.BitSet;

public class Primes {
  public int eratosthenes1(int numberToLookAt) {
    //no even number is prime so don't look
    // so the array only represents odd numbers
    int max = numberToLookAt/2;

    // remove the last even
    if ((numberToLookAt % 2) == 0)
      max = max - 1;

    //If bit i = 1 then the number (i*2)+1 is prime
    BitSet possiblePrimes = new BitSet(max+1);

    // Assume all the numbers are prime,
    // set them to true
    for (int i = 1; i <= max; i++)
      possiblePrimes.set(i);
```

1. See Appendix A for a list of machines and environments for timings in the book.

```
    int countOfPrimes = 1; // 2 is prime
    int currentBit = 1, step = 3, nextNonPrime = 4;
    while (nextNonPrime <= max) {
      if (possiblePrimes.get(currentBit)) {
        countOfPrimes++;
        int nonPrime = nextNonPrime;
        while (nonPrime <= max) {
          possiblePrimes.clear(nonPrime);
          nonPrime += step;
        }
      }

      currentBit++;
      step += 2;
      nextNonPrime += ((2 * step) - 2);
    }

    for (; currentBit <= max; ++currentBit) {
      if (possiblePrimes.get(currentBit))
        countOfPrimes++;
    }
    return countOfPrimes;
  }

  public int eratosthenes2(int numberToLookAt) {
    // no even number is prime so don't look
    // so the array only represents odd numbers
    int max = numberToLookAt/2;

    // remove the last even
    if ((numberToLookAt % 2) == 0)
      max = max - 1;

    //If bit i = 0 then the number (i*2)+1 is prime
    //Assume all the numbers are prime, already set
    BitSet possiblePrimes = new BitSet(max+1);

    int countOfPrimes = 1; // 2 is prime
    int currentBit = 1, step = 3, nextNonPrime = 4;
    while (nextNonPrime <= max) {
      if (possiblePrimes.get(currentBit) == false){
        countOfPrimes++;
        int nonPrime = nextNonPrime;
        while (nonPrime <= max) {
          possiblePrimes.set(nonPrime);
          nonPrime += step;
        }
      }

      currentBit++;
      step += 2;
      nextNonPrime += ((2 * step) - 2);
    }

    for (; currentBit <= max; ++currentBit) {
      if (possiblePrimes.get(currentBit) == false)
```

```
      countOfPrimes++;
  }
  return countOfPrimes;
}

public int eratosthenes3(int numberToLookAt) {
  // no even number is prime so don't look
  // so the array only represents odd numbers
  int max = numberToLookAt/2;

  // remove the last even
  if ((numberToLookAt % 2) == 0)
    max = max - 1;

  //If bit i = 0 then the number (i*2)+1 is prime
  //Assume all the numbers are prime, already set
  boolean[] possiblePrimes = new boolean[max+1];

  int countOfPrimes = 1; // 2 is prime
  int currentBit = 1, step = 3, nextNonPrime = 4;
  while (nextNonPrime <= max) {
    if (possiblePrimes[currentBit] == false) {
      countOfPrimes++;
      int nonPrime = nextNonPrime;
      while (nonPrime <= max) {
        possiblePrimes[nonPrime] = true;
        nonPrime += step;
      }
    }

    currentBit++;
    step += 2;
    nextNonPrime += ((2 * step) - 2);
  }

  for (; currentBit <= max; ++currentBit) {
    if (possiblePrimes[currentBit] == false)
      countOfPrimes++;
  }
  return countOfPrimes;
}

public static void main(String[] s) {
  int n = 1000000;
  Primes p = new Primes();
  int count = 0;

  long time1 = System.currentTimeMillis();
  count = p.eratosthenes1(n);
  long time2 = System.currentTimeMillis();
  System.out.println("eratosthenes1 found " +
                      count +
                     " primes in the first " + n +
                     " numbers in " +
                     (time2-time1) +
                     " milliseconds");
```

```
        time1 = System.currentTimeMillis();
        count = p.eratosthenes2(n);
        time2 = System.currentTimeMillis();
        System.out.println("eratosthenes2 found " +
                           count +
                           " primes in the first " + n +
                           " numbers in " +
                           (time2-time1) +
                           " milliseconds");

        time1 = System.currentTimeMillis();
        count = p.eratosthenes3(n);
        time2 = System.currentTimeMillis();
        System.out.println("eratosthenes3 found " +
                           count +
                           " primes in the first " + n +
                           " numbers in " +
                           (time2-time1) +
                           " milliseconds");

    }
}
```

After Coding

After you have written your code and it produces the correct functional results, you can test it to see if its performance matches the requirements you determined during design. Part 3 contains a more thorough discussion and examples of methods of benchmarking your own code.

It is likely that the code will not meet the performance requirements the very first time it runs. Even if it does, it may be worthwhile to look for some ways to improve it.

1.3 Types of Performance Problems

Figure 1-1 illustrates a useful approach to thinking about performance problems. As shown in Figure 1-1, there are three general levels of problems: those that are immediately obvious and easiest to deal with (low-hanging fruit), those that are the product of the application design, and those that stem from the “physics” of the situation—a term we use to refer to how things work, especially things that we can’t change directly. If we understand how things work, we can make better choices, make the best of what we have, or avoid problems.

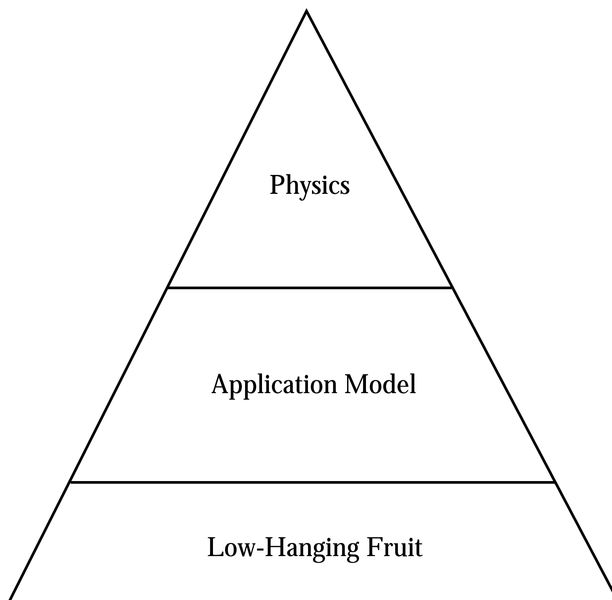


Figure 1–1 The Performance Pyramid

One interesting aspect of performance problems to keep in mind is that as you fix one set of problems, another set may present itself. It is often the case that one problem may create a bottleneck that masks all sorts of potential problems down the road. This causes the task of fixing performance to be iterative rather than a simple straight path. You will often find yourself revisiting each of the layers as you solve performance problems.

In addition, although you may find that the order of approaching performance problems can often be from the low-hanging fruit level to the top, you will need to start with the physics sometimes and with the application design at other times. The task at hand will usually indicate a starting point, as you'll see in the examples throughout the book.

1.4 Low-Hanging Fruit

Just as it is easiest to pluck an apple from the lowest branches of a tree, the first level of problems are usually the easiest to find and fix. The kinds of problems that are found in the low-hanging fruit level are often caused in the

implementation of the application. Problems here may range from fairly simple language misuse to outright bugs.

Even the best programmers make mistakes. The first problem is how to tell where the mistakes were made. Chapter 2 will go into more detail on how to locate where performance problems lie. The solution to these performance problems is often a simple substitution (a better algorithm, a more appropriate utility object) without changing the overall application structure or design. These problems can be hard to find but are easily fixed.

1.5 Application Design

The second level of problems originates with the application design. What may seem like a perfectly fine way of doing things at an abstract design level may turn out to have quite disastrous impacts farther down the line. Sometimes these problems can be fixed with a few minor tweaks of the design, but often the basic design must be changed.

The design of an application and how it interacts with all layers of the system can have a lot to do with its performance. This may seem rather obvious, but all too often performance issues are ignored or not understood and pieces (or the whole) of the design end up being redone. The traditional caveat of leaving performance considerations to the end is misleading. Performance issues do not need to dominate the design phase, but you do need to keep them in mind.

Poor Design Choices

Making the right choices at the right time is one of the keystones of a good design. This includes both data structures and logic. For example, if you are designing code that needs to use a collection, the choice of collection can be very important. Do your data need to be sorted? If not, then choosing a collection that keeps the data in a sorted state could be a big performance loss for no gain in needed function. Is the class you are designing accessed remotely? If so, then putting lots of little methods on the interface is probably a bad plan. Are your objects meant to be shared? If so, then you need to think about locking or synchronization.

One important design choice follows the suggestion mentioned earlier to "Never do today what you can put off until tomorrow." Suppose you need to insert objects into a shared persistent collection. After you have modified the

collection itself, you will have a database write lock on it, and no one else can use it. If there is work you can do before updating the shared object, then do that work first. For example, if you need to read data from other databases before you can complete updating the collection, then read them first rather than holding a write lock unnecessarily for a long time. Then, at the last moment update the shared object and finish the transaction. We will talk more about this kind of activity in Chapter 6, “Bottlenecks.”

Information Hiding

In order to make the right choice at the right time, the right information must be available. A common beginning to many discussions on how the design of the application missed its performance goals is something like, “I forgot that ...”, or “I did not realize that....” In some cases, information that is hidden from you causes the problem. “Data hiding” and “black box” are often used to describe the advantages of object-oriented programming. Sometimes what you don’t know (or forgot) can hurt you. The problem with what you don’t know is that sometimes you make important design decisions based on assumptions that are not true (see the Physics discussion that follows). Problems of this kind often require significant redesign to correct.

For example, consider an appointment calendar application (see Figure 1-2). The setup for this is a series of loops. An *EmployeeCalendar* class might contain a collection of *WorkWeek* objects that might contain days of the work week, Monday through Friday. These all get initialized at setup. That’s simple enough, but the application must account for national and local holidays that

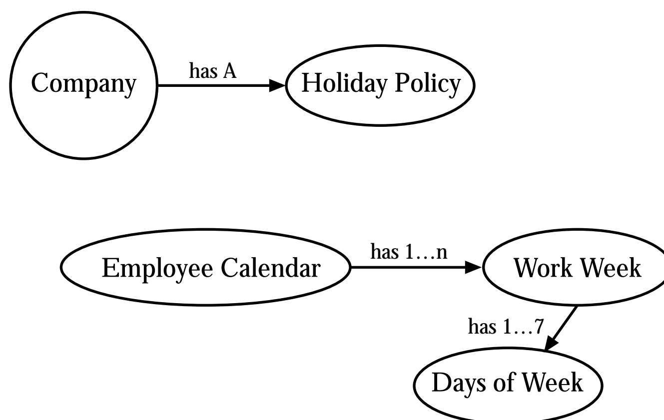


Figure 1–2 Employee Calendar

the company recognizes. Again, that's simple enough: as days of the week are added, use a method like *getActiveCompany()* to access the current company and select the holiday policy and check it as the application loops over the days of the week. So far, it appears that only the *WorkWeek* class needs to access the *Company* object. The *WorkWeek* contract may neglect to mention that it accesses the *Company*. This information hiding is justified because the use of the *Company* is an internal design detail of *WorkWeek* that could change later.

This initial design will work and may perform well enough in unit testing with a single server and a small (toy) database. When the application is deployed at a geographically distributed company though, the *Company* object is now on the headquarters server in the next state. The application now has a remote method call in the middle of the innermost loop of the employee calendar application.

This is a variation on the "constant calculations with a loop" problem mentioned earlier. The *Company* and its *Holiday Policy* are not constants in the usual sense, but for an employee calendar setup they are effectively constant. The design solution is to retrieve information from the *Company* and *Holiday Policy* before calendar setup begins and then to pass this information, as parameters, down through the application layers to the inner loop for *WorkWeek*.

This solution requires changes across several levels of the application. The *Company Holiday Policy* is accessed in a layer of the application that does not actually need access to the *Company Holiday Policy* data. This implies that the *EmployeeCalendar* should have known about internal details of the *WorkWeek* (the fact that it needed to check the *Holiday Policy*). This requires that method signature changes are needed for the *WorkWeek* class and any classes intermediate between *EmployeeCalendar* and *WorkWeek*.

1.6 Physics

The final layer of problems deals with those that form the underlying physics of the application and the environment it runs in. Physics is short for what you can't change but must understand to make the best of the current situation. This covers a range of elements, including:

- Communications factors such as the speed and physical components of the communications network, the bandwidth of IO channels, and the access speed of disk storage

- Requirements or limitations of purchased software, such as the database used for object persistence, how the garbage collector works for a particular Java Virtual Machine (JVM), or why Java is different from C++, COBOL, or RPG
- The overhead of switching environments and data transforms required between environments
- Economic constraints (you can use any computer system you want as long as it costs less than $20,000)
- Political factors (the Human Resource system that holds the employee data must be physically isolated and behind a firewall)

Just as a thorough understanding of physics is required to build rockets to send astronauts to the moon or build a transistor that works, a thorough understanding of Java Physics is invaluable to building enterprise-scale applications in Java. This understanding must extend beyond Java and include Java's interactions with operating systems, middleware, and distributed computing. This understanding allows architects and designers to make design tradeoffs and build workable systems.

Of course, no system works perfectly the first time, and Java Physics also applies to finding and fixing the problems encountered in these environments. Just as any good physicist learns how to develop a hypothesis and devise experiments that prove or disprove that hypothesis, the Java Physicist needs to develop similar skills. One of the most important skills is designing benchmarks and using them in a disciplined manner to isolate components and their interactions.

So even though good coding practices can avoid or fix some problems, other problems are in the design and result from a misunderstanding or violation of the rules of Java Physics. These problems are often hard to spot in the *development environment*[2] and only become visible at deployment time of the final application in the customer's environment.

The Java Language and Its Environment

Understanding the physics of Java also means separating myth from fact. Because Java is relatively new and its class libraries are changing rapidly, it is ripe for unfounded assumptions. Another important issue relates to how Java

2. The development environment is usually a single (uniprocessor) workstation and unit test involving toy databases. Scaling up the full databases, multiprocessor servers, and distributed hosts has a tendency to expose latent problems.

is being used. If Java is used as a direct replacement for an existing application written in an existing language, then a direct comparison between the old application and the new one is appropriate. However, Java has some unique capabilities:

- cross-platform portability
- multithreading support
- built-in networking Internet support
- built-in distributed programming support

all of which are bundled into a single (relatively) easy-to-use package. Exploiting these new features of Java not only makes it difficult to extrapolate from previous experience but often introduces new performance challenges. It is also difficult to compare the applications if the new application has more functionality than the old application. How much a new function is worth in terms of trading it for performance in other areas can be very tricky. If the new function is essential to a new requirement, then there really is no comparison.

The real question is, what is going on in your application environment? The reasons you selected Java for a project (those unique features described earlier) also require some design tradeoffs that are the fundamental reasons for the differences between Java and C++. For example, supporting threads as a fundamental part of Java requires a thread safe runtime library. Also, the pointer safety, garbage collection, and security features of Java require a more restrictive (and more object-oriented) environment than C++. There is no such thing as a free lunch. The many benefits of the Java environment have a runtime cost that must be understood and accounted for.

Interactions with Other Environments

Most interesting applications involve multiple middleware products, such as a Relational Database Manager (RDBM), network services, existing (non-Java) applications, or even an Object Request Broker (ORB), each with its own environment. The physics involved are the overhead of transporting requests and information between environments. The cost of leaving the Java environment and entering the environment of target service (and vice versa) must be understood and accounted for in any performance analysis. For modern Java[3] this starts with the cost of a Java Native Interface (JNI) call.

3. Java 1.0 supported the Native Method Interface (NMI), which was found to be incompatible with Java's cross-system portability goals. JNI was introduced with Java 1.1.

Moving in and out of the Java environment is not the only issue in working between environments. For example, to retrieve a row from an RDBM, Java calls native code (via JNI), which will invoke another Application Program Interface (API) such as Microsoft's Open Database Connectivity (ODBC), DB2's Call Level Interface (CLI), or Oracle's Oracle Call Interface (OCI). These APIs are supported by interface mapping code that then communicates with the RDBM (yet another environment switch) and then returns to the native code and finally back to Java. Each of these environment switches entails significant overhead for each transition (in and out).

We like to think of these environments as a series of valleys with high mountains between the valleys. There are "passes" over the mountains to get from one valley to another (APIs like JNI and ODBC). It takes more energy to get over the pass than to drive around on the valley floor. The cost (in time and fuel) for driving over the mountains to the "next valley" is high enough that you want to plan your trip carefully (making an extended shopping trip, not just a run for milk and eggs).

There is also a significant cost to transform data to and from Java native format. This is the cost associated with moving in and out of the single Java language environment. This is the cost of converting Java objects fields to a flattened form (object stream) for sending over a network interface, to a form usable by native code, or to a form compatible with the fields of a relational database.

Strings are an obvious case. In Java, strings are objects encoded in Unicode format. Other environments (C, C++, RDBMs) store strings in different (nonobject) formats and encoding (ASCII, EBCDIC, and specific code pages). These transformations often require additional overhead in the form of up-calls into Java. For example, if a native method returns an ASCII text string to Java, this involves an up-call into Java to create and initialize the string object.

Interacting with other environments brings a number of implications to bear. First, the cost of environment switching across APIs and data conversion at API boundaries can be significant. It is not always obvious that moving a function from Java to native code will improve performance. In fact, the cost of calling through JNI and the required data conversion may exceed the benefit of moving the function. With modern Just In Time (JIT) techniques or even direct compilation, it usually pays to stay in Java where possible.

Second, the cost of accessing and converting data to Java native form can be extremely high. This creates a strong imperative to cache the transformed data as Java objects for time. It is also especially useful for new Java applications where there is no requirement for concurrent access from non-Java applications.

Persisting Java Objects

The cost of transformation is magnified when the task is to transform a complete Java Object into a row of a relational table and back again into a fully functioning Java Object. The degree of difficulty is increased because an object is more than the sum of its fields. An object instance requires additional runtime information (metadata) to support its object-oriented features: class, class hierarchy, polymorphism, and references to other objects, each with its own metadata. This requires the storage of additional information (metadata) beyond the object's fields in the database. This information (object identity, class identity, object and class identity of referenced objects, and so on) can be substantial and can exceed the size of the object's fields by themselves.

This is just the beginning of complex issues involving the mapping the contents of an arbitrary Java Object into a specific field layout (or schema) of a relational table. We call this schema mapping for short. The simplest form of schema mapping an object to a relational table is to stream the object and any objects it owns (the transitive closure) and save this stream as a single long column in the relational database (RDB).

In its simplest form, streaming an object involves converting the object's metadata and each object field to a string of bytes and laying them end to end into one large array of bytes. This array of bytes can then be written over a wire or into a field of a relational table. This description should give the reader some sense of the computation cost of object streaming. In its simplest form, object streaming involves at least one method call per field.

Another issue is that the object's representation grows in size when streamed. In the general case, just stacking the fields end to end is not enough to ensure the correct recovery of a Java object from the streams. Each object and/or field must be identified so that the code reading the stream can recover the data correctly. So each *int* field must be preceded by a byte indicating that the following four bytes represent an *int*.

In Java, anything that is not a primitive type (like arrays, strings, and other objects) is implemented as a reference to an object. Arrays of primitives and strings can be represented in a simple form (type identifier, length, and contents), but arrays of objects require a more complex representation. As arrays of objects are typed in Java, the representation must include the fully qualified class name of the elements of the array in addition to the length and streamed contents of each referenced object.

For each referenced object in the current object, the stream must contain the fully qualified class name (or its equivalent) in addition to the contents of

the object itself. This is required to support normal object-oriented behavior (polymorphism), but it places into the streamed form of the object quite a lot of extra data that were not there in the runtime form, for example,

"com.mycompany.myproduct.mycomponent.myLongClassName"

for each object instance in the stream. This process continues recursively until all objects reachable from the *root* object are traversed. In our experience, an object stream contains as much metadata as instance data and sometimes more. This is also true for the streaming data across a communication link.

Why is this important? Simply put, bigger is slower. If every record is bigger, then it naturally requires more CPU cycles to copy data between buffers and more time to read and write records to disk. With larger records, fewer records fit into the RDBM's memory cache, requiring more disk IO for a given workload.

This sort of streaming is the most general and simplest to implement case. The Java runtime does not know the "programmer's intent" and must assume that an object reference may contain an instance of any subclass or subinterface. To reduce the overhead of persistence requires more information from the environment and even the developer.

The alternative is to use streaming only for exceptionally complex cases and to allow the programmer or database administrator to specify a specific mapping for each field. This can be extended to include specific programmer knowledge. For example, *java.lang.String* can be an arbitrary length, but the programmer knows that the actual length does not exceed 25 characters. A schema mapping tool may allow the programmer or database administrator to tell the schema mapping runtime that a specific field is always an instance of a specific class.

For example, a reference to a *PostalAddress* may be an instance of a *PostalAddress* or any valid subclass of *PostalAddress*. If the programmer knows that, in this case, the reference is always exactly an instance of *PostalAddress*, then it is possible to define a mapping where the fields of *PostalAddress* are expanded as fields in the record of the containing object.

This knowledge allows for the possibility of removing some metadata from the record or object stream. This metadata must still be maintained either in the schema mapping tools metadata or directly in the implementation of code that implements the mapping. This can reduce the amount of persistent (meta)data to be written.

However, this tradeoff must be carefully considered. It may be true at the moment that a specific field always references instances of class *myClass*. But

what will happen next year? Eliminating metadata may make it difficult to support a future requirement. Because the metadata is not in the persistent form of the object, enabling full polymorphic behavior will require a mass conversion of one or more tables. Essentially this is trading off future advantages of object-oriented programming for a near-term performance gain.

The persistent discussion so far assumes that the persistent form of the objects will be stored in rows of relational tables. This works best if the data originated that way and the application makes limited use of object-oriented features. The results are less so when the application is new and fully object-oriented. It turns out that a relational database is a poor place to store some objects. The runtime form of Java objects and the stored form in the relational table are very different. This difference is attributable to what cannot be stored in a relational database: efficient references (pointers) and classes (code and pointers to code).

An alternative to storing objects in relational databases is to build a database specifically designed to store objects, so-called object-oriented databases (OODB). OODBs change the physics of the problem by making the stored form of an object more like the runtime form. This reduces or eliminates most of the data conversions and data expansion described earlier.

Time, Distance, and Space

The volume of data (space) and the distance it has to travel limits the performance (time) of any distributed system. Moving a small object a short distance is always faster than moving a large complex object a long distance. Better yet, don't move the object at all!

This is an easy thing to ignore with technology delivering ever-increasing bandwidth and capacity. The physics dictates that accessing an object in the on-chip cache of a microprocessor is faster than accessing the same object over 3 meters of gigabit fiber optic cable. Simply put, distance and the speed of light still matter. Add to this any buffering delays, interface, or protocol conversion overhead, and the result is nowhere near the speed of light.

This implies that local/remote transparency (not being able to tell if an object is local or remote) is never really "transparent." It may be transparent from a programming perspective, but not from a performance perspective. A method call via a stub is slower than a direct method call, and it is slower yet if the stub streams data over a socket (even the local loop-back socket). We will cover this in much greater depth in Chapter 4, "Local/Remote Issues."

The reality is that locality of reference is still important at all levels of the system and application. The design problem is to ensure that objects that

interact frequently are physically near and that less frequently accessed objects are remote as dictated by other design constraints or requirements.

Of course, some performance issues are common to all business computing. A well-tuned database is still important. The design of the I/O subsystem (number and speed of the channels, number and type of the disks) is still important. Having a computer system designed as a server for business computing (scalable memory and caches, symmetrical multiple processor (SMP) configurations, direct memory access (DMA) speed, and I/O attachment capacity) is still important.