

Building High-Performance Applications and Servers in Java: An Experiential Study

Sandeep K. Singhal

IBM T.J. Watson Research Center

Binh Q. Nguyen

Michael Fraenkel

Richard Redpath

Jimmy Nguyen

IBM Software Solutions

Building High-Performance Applications and Services in Java: An Experiential Study

Sandeep K. Singhal

IBM T.J. Watson Research Center

Binh Q. Nguyen

Michael Fraenkel

IBM Software Solutions

Richard Redpath

Jimmy Nguyen

Abstract

The Java programming language is rapidly becoming the language of choice for developing dynamic content for the Internet. The language's popularity has arisen from its portability, ease-of-use, and integration with HTML. Java is being used to enable animation on Web pages, to dynamically select and format Web page content at Web servers, and to provide client-side user input checking as a front-end to transaction-oriented applications. However, with a few exceptions, Java has not been used to develop applications and servers that demand high performance or throughput.

In this paper, we describe techniques for improving the performance of Java code and enabling the development of high-throughput applications and servers. We show that Java code can be easily optimized to achieve performance that is comparable to code developed in traditional languages such as C or C++. We base these results on our experience in developing the InVerse (Interactive Universe) server in Java.

Caveats

Practitioners who read or use the results in this paper should bear some things in mind:

- **We have only performed tests on Sun's 1.0.2 Java class library and virtual machine on one platform.** Other hardware, Java virtual machines, and class library implementations are unlikely to exhibit identical behavior, though we expect that many of the described results are universally applicable.
- The numbers presented in this paper were generated using a millisecond-resolution clock that takes up to 0.5 milliseconds to execute. As a result, **our numbers may exhibit a relatively large margin of error.** Differences of 10-20% can safely be attributed to statistical error, garbage collection effects, or clock resolution. We have therefore focused our attention on performance differences that have greater significance.
- **This study is by no means exhaustive. It represents analyses performed to support a particular server application.** Though we expect the results to be of general use, we have not assessed whether the operations common in our application are equally common to other applications.

Conclusions: Improving Java Application Performance

Java application performance can be improved by applying the following optimizations:

- Eliminate synchronization in application code. Implement unsynchronized versions of Java library classes and use those versions when the application is single-threaded or data structure access controlled by design.
- Determine the most common operations performed on standard Java classes and re-implement those classes to speed up those common cases.
- Merge Java classes when interactions can be optimized
- Where errors are rare, avoid checking for errors and instead catch exceptions (**ArrayIndexOutOfBoundsException**, **ClassCastException**, etc.) generated by the Java run-time.
- Avoid creating temporary objects and, where possible, re-use objects by providing a **reinitialize()** operation.
- Limit variable reads and writes where possible. Cache class static variables and instance variables in method stack variables when possible because the latter provide faster access. Reverse the order of common loops to count down to zero instead of counting up to a variable limit.

Building High-Performance Applications and Services in Java: An Experiential Study

Sandeep K. Singhal

Binh Q. Nguyen
Michael Fraenkel

Richard Redpath
Jimmy Nguyen

IBM T.J. Watson Research Center

IBM Software Solutions

Abstract

The Java programming language is rapidly becoming the language of choice for developing dynamic content for the Internet. The language's popularity has arisen from its portability, ease-of-use, and integration with HTML. Java is being used to enable animation on Web pages, to dynamically select and format Web page content at Web servers, and to provide client-side user input checking as a front-end to transaction-oriented applications. However, with a few exceptions, Java has not been used to develop applications and servers that demand high performance or throughput.

In this paper, we describe techniques for improving the performance of Java code and enabling the development of high-throughput applications and servers. We show that Java code can be easily optimized to achieve performance that is comparable to code developed in traditional languages such as C or C++. We base these results on our experience in developing the InVerse (Interactive Universe) server in Java. Our results are applicable to Sun's Java class library and virtual machine, but other implementations may exhibit other performance characteristics.

1 Introduction

The Java programming language is rapidly becoming the language of choice for developing dynamic content for the Internet. The language's popularity has arisen from its portability, ease-of-use, and integration with HTML. Java is being used to enable animation on Web pages, to dynamically select and format Web page content at Web servers, and to provide client-side user input checking as a front-end to transaction-oriented applications. Java has also emerged as the language of choice for scripting 3-D motion in Virtual Reality Modeling Language (VRML) [HW96] worlds and for enabling multi-user interactions within these virtual environments [VRML96].

Despite all of these established uses for Java, the language has rarely been used to develop applications and servers that demand high performance or throughput. One of the primary reasons cited by developers for not using Java in these situations is a perceived inability to write Java code that executes at speeds competitive with traditional compiled languages such as C or C++. Sun's

own performance measurements indicate that interpreting Java bytecodes degrades application performance by between a factor of two and a factor of twenty [VH95]. Limited experience with garbage collection and JIT (Just-In-Time) compilation in large-scale applications are further factors that affect developers' comfort with Java performance.

Although these concerns have a level of validity, we have found that they may be successfully overcome by employing a combination of careful program design and targeted optimization. In this paper, we describe techniques for improving the performance of Java code and enabling the development of high-throughput applications and servers. We show that Java code can be easily optimized to achieve performance that is comparable to code developed in traditional languages such as C or C++. We base these results on our experience in developing the InVerse (Interactive Universe) server [S+97] in Java.

All of our performance measurements were generated on an unloaded 120MHz Pentium processor machine with 48MB of memory and running Windows 95. We used the Java 1.0.2 compiler and virtual machine provided by Sun with the `-O` flag to signal optimization. Timings were generated by executing the target code block at least 50,000 times within a tight loop and by using the `System.currentTimeMillis()` call to retrieve (wall clock) timing information. We also executed our test cases using a Just-In-Time (JIT) code generator that produces native instructions from the Java byte codes at run-time.

The next several sections describe the primary performance issues faced by the Java application developer, along with approaches for optimizing those performance problems:

- Excessive synchronization
- Excessive memory access
- Excessive object creation and garbage collection
- General-purpose class library
- Excessive error checking

	Unsynch.	Synch.
Without JIT		
No thread contention	1.70	16.12
Thread contention	3.94	67.45
With JIT		
No thread contention	0.36	18.82
Thread contention	0.72	69.95

Table 1: Method Call Time (in Microseconds) With Various Combinations of Synchronization and Thread Contention

Although many of these issues are applicable to any object-oriented programming language, they are particularly germane to Java developers. We conclude by describing how optimizing these areas improved the overall performance of the InVerse server application and discussing our future work in this area.

2 Excessive Synchronization

In Java, class methods may be marked with the **synchronized** keyword, which ensures that multiple threads cannot simultaneously be executing within any of the synchronized methods on the same class instance. To provide this functionality, the Java virtual machine attaches a *monitor* to each object having synchronized methods; whenever a thread attempts to enter a synchronized method, it must first obtain a lock on the monitor for the particular object being referenced. The developer may also attach the **synchronized** keyword to a code block, at the same time specifying a particular object on which synchronization should be performed. The Java virtual machine treats entry to a synchronized block much like entry to a synchronized method on the associated object.

As shown in Table 1, accessing a synchronized method can slow application execution speed by up to nearly a factor of 100. The table shows the time required to call a zero-line function on an object with various combinations of method synchronization and thread contention. In the case of thread contention, two threads are simultaneously executing the same tight loop trying to call the object's method.

The table reveals that adding synchronization to a method degrades performance by a factor of between 9 and 50 *even if the application is only executing a single thread* (i.e. the thread would never block on the monitor). This result indicates that the synchronization overhead is primarily incurred in checking for the lock availability, rather than in actually obtaining the lock.

Comparing the synchronized method call overhead for the non-JIT and JIT cases, we see that the JIT compiler is unable to optimize the lock access and contention at all. Indeed, the performance overhead of

	Unsynchronized	Synchronized
Without JIT		
byte	0.52	1.00
char	0.80	1.72
short	0.85	1.79
int	1.37	3.15
long	2.86	6.27
float	1.45	3.24
double	2.80	6.28
With JIT		
byte	0.08	0.68
char	0.14	1.13
short	0.13	1.06
int	0.23	1.94
long	0.49	4.23
float	0.27	2.04
double	0.55	4.20

Table 2: Effect of Synchronization on **DataOutputStream** Performance (Normalized to Cost of Outputting a Single Byte Using Standard Synchronized Implementation)

the JIT compiler actually slows down the program execution. In contrast, the JIT compiler improves the overhead by a factor of five or more when the calls are unsynchronized.

Finally, the numbers reveal that thread switching is relatively expensive. With two threads running simultaneously, the total function call throughput actually decreases by more than a factor of two. We suspect that this overhead is caused by synchronized method calls within the Java thread implementation.

The cost of synchronization can impact application performance in a number of subtle ways. For example, the memory allocator is synchronized, meaning that object creation incurs a synchronization penalty. Furthermore, many common functions in the Java class library are designed to be thread-safe and are, therefore, synchronized.

For example, accessing an indexed element in a **Vector** requires a synchronized method call, as does calling the **nextItem()** on an associated **Enumeration**. In our experience, many collections are only accessed by a single thread, so this synchronization offers no benefit and imposes considerable overhead. To allow the application to selectively disable this synchronization, for example, we have subclassed the Java **Vector** class to provide an unsynchronized access method.

Similar synchronization issues arise in the I/O stream library. Writing formatted data through a **DataOutputStream** calls a synchronized **OutputStream** method to output each byte of data. For example, writing a long integer to a stream incurs the overhead of eight synchronized method calls. Table 2 shows the overall

performance of the **DataOutputStream**. The table compares the standard (synchronized) version against a version with synchronization removed. The table reveals that synchronization accounts for up to 86% of the overhead for outputting data. Furthermore, while the JIT compiler can improved synchronized performance by only 33%, it can improve the performance of the unsynchronized class by up to 84%.

We have found that where synchronization cannot be eliminated, we can occasionally improve performance by placing a series of synchronized calls inside a single synchronization block. For example, we can replace

```
for (i=0; i<count; i++)
    x.f();           // synchronized on x
```

with

```
synchronized(x) {
    for (i=0; i<count; i++)
        x.f();       // synchronized on x
}
```

The replacement code allows the Java virtual machine to short-circuit the repeated synchronization points. It should only be used if the synchronization is short-lived (because it will otherwise block other threads) or if there is no thread contention. Unfortunately, Java's caching of monitors causes this code to exhibit variable performance on each execution.

3 Excessive Memory Access

With the increasing divergence between processor speeds and memory access times, high-performance application developers are acutely aware of the need to minimize memory access. In compiled languages such as C++, the access time to all variables is fairly uniform because their memory locations are pre-computed, or, in the worst case, accessed through one or two levels of indirection. However, Java application performance is noticeably affected by what types of variables are accessed and how they are accessed. For example, while stack variables are directly addressable (and may even be placed in registers), instance variables typically require an extra level of indirection to be accessed.

Table 3 illustrates the overhead for accessing local method (stack) variables, instance variables within the current class and superclasses, and static class variables. The table shows that reading local method variables is about 30% faster than accessing class instance variables. With JIT, the difference is even more significant. The table also indicates that writing instance variables in parent classes is faster than accessing instance variables in derived classes. The variations in read access times are small enough to indicate no conclusive pattern.

	Read	Write
Without JIT		
Local method variable	1.00	1.00
Object instance variable	1.49	1.32
Superclass instance var.	1.40	1.32
Supersuperclass inst. var.	1.53	1.26
Class static variable	1.81	1.21
With JIT		
Local method variable	0.69	0.00
Object instance variable	1.59	0.16
Superclass instance var.	1.62	0.13
Supersuperclass inst. var.	1.65	0.07
Class static variable	1.94	0.20

Table 3: (Normalized) Access Times for Different Variable Types and Modes

This memory access pattern implies the potential value of dynamic *data location shifting*, dynamically changing the storage location of data based on the access patterns. For example, a data-intensive operation would benefit from first copying instance variables into stack variables, operating on the stack variables, and, finally, copying the stack variables back to the permanent instance variables. We have found this technique to be particularly useful when a method instance variable is accessed repeatedly within a loop. Furthermore, the cost of memory access means that the common loop construct:

```
for (int i=0; i<limit; i++)
```

can be improved by 21% (5% with the JIT compiler) by rewriting it as

```
for (int i=(limit-1); -1 < --i; )
```

to reduce the number of accesses to the **limit** variable.

Memory access behavior can manifest itself in subtle ways. For example, consider the task of testing the bits of a 32-bit value. A standard approach would loop 32-times, shifting the value by one each time and testing the right-most bit. However, as shown in Table 4, it is generally faster to shift fewer times and check multiple bits (using a series of if-then statements and various bit-masks) on each iteration. This result is explained in part because this loop unrolling reduces the overhead for loop bounds checking. This performance improvement is further explained because the if-tests reduce the number of shift operations required on the data. A shift costs over 8 microseconds, which is 35% more expensive than

	With -out JIT	With JIT
32 shifts by 1, one test each	1.00	0.59
16 shifts by 2, two tests each	0.63	0.28
8 shifts by 4, four tests each	0.45	0.15
4 shifts by 8, eight tests each	0.32	0.10
2 shifts by 16, sixteen tests each	0.26	0.07
thirty-two tests	0.11	0.00

Table 4: (Normalized) Performance of Various Bit-Test Strategies

a local variable write and 250% more expensive than a local variable read.

4 Excessive Object Creation and Garbage Collection

Armed with Java's built-in garbage collection, programmers find themselves free to no longer worry about how memory for their objects is allocated and whether it is explicitly released. Although garbage collection considerably simplifies the task of object-oriented development, its availability can also cause significant performance problems within Java programs. Programmers are prone to create objects liberally because they perceive objects to be relatively cheap to create.¹ In effect, the Java virtual machine masks the costs of memory allocation and deallocation for objects.

Unfortunately, object creation and destruction in Java can be extremely expensive, particularly in applications that repeatedly create transient objects. Table 5 shows the execution times for a variety of object creation operations; these values are compared to the expense of extracting an item from an array, as might be done within a free object pool. The numbers reflect the worst-case performance of the array access (using synchronization to ensure consistency) and the best case performance of object creation (shallow class hierarchy with no instance variables). The table reveals that object creation is almost always more expensive than the alternative of re-using existing objects. For accessing the object repository, the JIT performance is poor primarily because of the synchronization overhead illustrated in Table 1. In our experience, many objects are created at a well-defined point in the program, thereby eliminating the need for synchronized access to the object repository and, consequently, improving performance by a factor of 16.

	With- out JIT	With JIT
Create object (no constructor)	11.55	8.36
Create object (constructor)	11.98	8.35
Create subclass (no construct.)	14.73	10.16
Create subclass (base class constructors)	16.87	9.84
(Synchronized) remove element from array	10.94	9.28

Table 5: Execution Time to Create Java Objects (in Microseconds)

Our object creation measurements reveal an important tradeoff between constructor complexity and subclassing. In general, object creation gets slower as the inheritance hierarchy grows deeper. At the same time, it is generally cheaper to execute a constructor than to work with a deeper subclass. This result suggests that one must be careful about specializing for minor variations that could otherwise be represented by a (constructor-initialized) state variable. As an added benefit, reducing the depth of class specialization improves the load time of the applet because fewer classes must be transferred across the Web. Clearly, changing the object hierarchy might sacrifice the object-oriented design of the application, so it must be done with care.

Excessive object creation can have more significant effects on compute-intensive applications such as servers. The Java garbage collector is supposed to only execute when no other application threads are executing; consequently, the garbage collector should have minimal effect on an application's performance. However, because high-performance applications may not offer such idle periods, garbage collector performance must inherently steal time from the application execution.

The negative effects of memory allocation overhead appear throughout the Java class library, for instance. When a number is passed to an output stream for display, the Java library creates dozens of objects representing different portions of the number as instances of **Character** and then as instances of **String** [d96]. Furthermore, when transmitting and receiving datagram (UDP) packets, the class library forces the programmer to create transient instances of **DatagramPacket** for each packet.

To reduce the number of objects created and destroyed, we have employed two techniques:

- When only one instance of the object is needed at a time (e.g. within the body of a function), then we

¹ Contrast this behavior with that of C++ programmers who are acutely aware of the costs of **malloc()** and **free()** calls to allocate and free heap memory. To minimize these overheads, they often maintain customized memory heaps, replace the system **operator new** and **free()** calls to control placement of objects within those heaps, and maintain free object pools that allow objects to be reused without re-invoking a constructor.

declare a static variable to hold the object instance between successive function iterations. Within the function, a **reinitialize()** function is invoked on the re-used instance so that it may be treated as a newly allocated instance.

- When multiple instances of the object are active simultaneously and their lifetimes cannot be localized to a single part of the application, we use a **Vector** or growable array to implement a free object instance repository. When an object is no longer needed, a static method on its class is invoked to add the instance to the free object pool. Similarly, when the application needs a new instance of the class, it invokes another static class method that releases an available instance from the repository and invokes the **reinitialize()** method on it. We declare the constructors as private to ensure that the application does not accidentally allocate new instances directly.

Both of these techniques require the presence of a **reinitialize()** method in the optimized class. On occasion, we have found the need to optimize Java library classes that do not supply this method and are declared **final** (thus preventing subclassing). In these cases, we have provided the **reinitialize()** method by implementing replacement versions of such classes such as **DatagramPacket**. We force the Java virtual machine to load our new implementation by placing its .class file earlier in the class search path.

5 General-Purpose Class Library

The standard Java class library was written to be usable by a broad variety of applications, ranging from simple Web applets to more complex systems. This generality manifests itself in several ways that are detrimental to high-performance server development in Java. Typically, performance problems arise when the application uses the provided class in a limited way or specialized context, thereby enabling specializations (and hence performance optimizations) that the library developers could not justify within the general-purpose library.

In this Section, we discuss two types of optimizations that we have performed on the Java library: optimizations based on class access patterns and optimizations based on class interactions.

5.1 Optimizing for Class Access Patterns

The Java libraries occasionally assume that applications will use those classes in a particular way. For example, the library developers may have optimized certain class methods at the expense of others. If the application predominantly calls the unoptimized methods, then

performance can be improved by shifting the class implementation to suit the application.

In Section 2, we described one example of this optimization to eliminate unnecessary synchronization. Most of the Java library was designed for multi-threaded applications and is, therefore, thread-safe, meaning that data structure manipulation functions are synchronized. Though this safety is generally desirable because it frees developers from worrying about thread interactions, it is not universally required. In our experience many data structures within servers are only manipulated by a single thread, and therefore, this synchronization can be selectively eliminated to improve performance.

We discovered another area of optimization when studying the performance of the **HashTable** data structure. The basic **HashTable** implementation, illustrated in Figure 1a, is optimized to enable fast lookups of entries based on their search keys. A hash code is computed for the search key, and that key is used as an array index to locate a linked list of entries within which the key is searched. Insertion and deletion are also relatively fast because those operations rely on the same key-value lookup operation.

In analyzing our application, we discovered that though we used the **HashTable** frequently for key-value lookup, we employed insertion and deletion relatively rarely. Instead, we frequently iterated through all values in the data structure (by obtaining an Enumeration through the **values()** method). Referring again to Figure 1a, we see that this iteration can be slow because as we reach the end of each linked list, the **Enumeration** must search along the main array to find the next occupied slot bound to a list.

We therefore implemented a specialized **HashLink** class that provides fast lookup and fast iteration, at the expense of slower insertion and deletion. As shown in Figure 1b, the **HashLink** retains the basic structure of a **HashTable** but also links all of the individual linked lists together into a single double-linked chain. A flag indicates whether each entry represents the tail of its

	ByteArrayOS/ DataOS	ByteArray- DataOS	SynchByte- ArrayDataOS
Without JIT			
byte	39.00	9.70	25.70
char	67.10	16.40	30.10
short	69.80	15.30	30.00
int	123.00	25.70	41.20
long	244.50	56.40	70.70
float	126.40	32.80	47.70
double	245.00	62.50	79.00
With JIT			
byte	26.40	1.10	20.90
char	44.00	3.30	20.80
short	41.20	3.30	22.00
int	75.80	4.40	21.90
long	164.80	17.00	34.60
float	79.70	7.10	24.20
double	163.70	15.90	37.40

Table 7: Performance of Java **DataOutputStream** on Byte Arrays Compared to Optimized Unsynchronized and Synchronized Classes

particular sublist. As shown in Table 6, execution time for the key-value lookup process is nearly identical to that seen by the **HashTable** because the lookup process is unchanged. However, **HashLink** iteration now simply involves traversing the primary linked list and is, therefore, faster. Though **HashLink** insertion and deletion involve more work than the corresponding **HashTable** operations, the resulting performance differences are not significant.

5.2 Optimizing Class Interactions

Like any good object-oriented library, the Java library relies heavily on interfaces that mask the underlying class implementation. Although this approach leads to modularity, it also limits overall library performance. For example, because classes must interact with each other through a general-purpose interface, they cannot take advantage of capabilities that are unique to their particular implementations. In cases where particular classes are interacting through such an interface, performance can be improved by specializing the interaction point.

An example of this specialization arises in the interaction between the **DataOutputStream** and **ByteArrayOutputStream** classes. **DataOutputStream** serializes high-level Java built-in types (int, long, float, double, **String**, etc.) to an object exposing the **OutputStream** interface. However, the **OutputStream** interface only allows the output of one byte at a time (via a synchronized method) because it is used to support a variety of output media including files, TCP/IP

	HashTable	HashLink
Without JIT		
Key-value lookup	96.60	99.00
Iteration	26.40	4.40
Object insertion	372.40	308.60
Object deletion	107.60	148.40
With JIT		
Key-value lookup	38.40	34.20
Iteration	8.80	1.20
Object insertion	167.00	117.60
Object deletion	40.80	43.80

Table 6: Performance of 50,000-Element **HashLink** Relative to **HashTable** (in Microseconds)

connections, and memory buffers. As its name suggests, the **ByteArrayOutputStream** class encapsulates the writing of bytes into a byte array.

When the **DataOutputStream** is attached to a **ByteArrayOutputStream**, performance is sub-optimal because data is only produced one byte at a time (through the **OutputStream** interface). To output each byte, the **DataOutputStream** incurs the overhead of calling a synchronized method, checking for array overflow, copying the output byte into the byte array, and incrementing the output byte counter.

We implemented a **ByteArrayDataOutputStream** class that merges the functionality of the **DataOutputStream** and **ByteArrayOutputStream** classes. Knowing that it is placing data into a byte array buffer, our optimized class simply copies the data into the buffer, checking the array bounds and incrementing the byte counter only once for each data item output. In keeping with our strategy of not imposing synchronization on applications that do not require it, methods in our optimized class are unsynchronized. To support applications that require synchronized access to the stream, we produced a subclass **SynchByteArrayDataOutputStream** that simply wrappers each of the data output methods into a synchronized method. Table 7 illustrates that our optimized classes perform up to 25 times faster than the standard Java class library.

We have employed the same technique to optimize the performance of **ByteArrayInputStream** and **DataInputStream** interactions and seen similar benefits.

5.3 Other Observations

Early in our optimization, we discovered that the **System.currentTimeMillis()** call is itself a rather expensive operation, requiring 0.51 milliseconds without the JIT compiler. The JIT compiler only improves performance to 0.49 milliseconds, most of which can be attributed to optimizations in the loop construct within

our test case. We suspect that the cost of this function arises because it is implemented as a call to the Windows operating system. As a result of this discovery, we eliminated many of our calls to this function, relying on cached time values whenever small time inaccuracies were tolerable.

The performance of `System.currentTimeMillis()` reveals one obvious limitation of JIT compilers. If a method is implemented as a **native** call to a C/C++ library, the JIT compiler can offer no further improvement, even if run-time optimizations might have otherwise been possible. The tradeoff between native calls (implemented by statically optimizing C/C++ compilers) and Java code (implemented by dynamically optimizing JIT compilers) remains an area of open research.

6 Excessive Error Checking

The Java virtual machine provides native instructions to support exception handling [LY97], in contrast to languages such as C++ which must rely on compiler-generated instructions to explicitly store exception information on the program execution stack and to locate the appropriate exception handler when an exception is thrown. The implications of this virtual machine support is that exception handling in Java is relatively fast. As shown in Table 8, a **try-catch** clause imposes negligible overhead as long as the called function does not actually throw any exceptions. The cost of actually handling a thrown exception is non-trivial, but that expense is only incurred in the error condition signalled by the exception.

The near-zero cost of **try-catch** clauses leads high-performance application developers to rely on exception handling instead of explicit error checking when errors are expected to be rare (i.e. the exception would rarely be thrown). Consider, for example, the task of indexing into an array:

```
if ((idx >= 0) && (idx < array.length))
    x = array[idx];
else
    // error
```

An exception-oriented approach leads us to re-write the above code block in the following form:

```
try {
    x = array[idx]
}
catch (ArrayOutOfBoundsException e) {
    // error
}
```

The exception-oriented approach is 70% faster than the traditional approach in the (common) case that the index

	Without JIT	With JIT
Function call	4.67	1.26
Function call in try-catch (no exception thrown)	4.62	1.10
Function call in try-catch (exception thrown)	8.30	4.98
If-test array index bounds	1.16	Test
Array dereference	0.52	Crashes
Array dereference throwing bounds exception	242.78	(JIT Bug)
If-test using instanceof	5.55	2.80
Object type cast	2.75	1.10
Object type cast throwing casting exception	259.00	274.60

Table 8: Overhead of Java Exception Handling (Microseconds)

is within range. As long as the common case occurs sufficiently often relative to the exception condition, we achieve a performance improvement by eliminating the error check on each execution.

A similar situation arises with type casting. For example, we can replace the standard type casting logic:

```
if (anObject instanceof MyClass)
    ((MyClass)anObject).func();
```

with the following code:

```
try {
    ((MyClass)anObject).func();
}
catch (ClassCastException e) {
    // error
}
```

The performance characteristics of **try-catch** clauses lead us to re-define how exceptions are used in high-performance applications. Instead of simply using exceptions to signal errors, we use exceptions to signal uncommon occurrences, even if those occurrences do not represent error conditions. This approach [C96] is consistent with “making the common case fast.” Consider once again the task of iterating through an array. For sufficiently large arrays (roughly 200 elements), we can replace the traditional code:

```
for (idx=0; idx < array.length; idx++)
    x += array[idx];
```

with the following exception-oriented approach:

```
try {
    for (idx=0; ; idx++)
        x += array[idx];
}
catch (ArrayOutOfBoundsException e) {
    // Iteration complete
}
```

}

7 Conclusion and Future Work

In developing our high-performance server, we have found the need to apply several specific optimizations that account for unique performance characteristics of the Java virtual machine and run-time system:

- Eliminate synchronization in application code and by re-implementing standard Java classes
- Develop customized versions of standard Java classes to match the access patterns exhibited by our application
- Merge Java classes when interactions can be optimized
- Selectively use exception handling to create a fast-path for common case execution
- Re-use object instances where possible and eliminate creation of transient instances
- Cache object variable references in stack variable references where possible

Although we had originally written the InVerse server with an eye toward good performance, these optimizations enabled us to further improve performance by between a factor of eight (for the least common cases) and a factor of 100 (for the common case code paths), as shown in Table 9. Based on our measurements, we expect that the JIT compiler will further enhance the impact of these optimizations on the InVerse system performance.

Probably most important, these impressive performance improvements did not require significant changes to the application's basic design. We did not have to compromise the object-oriented structure of our application or destroy the encapsulation of its various classes and components. Indeed, each optimization typically involved modifications that were localized to within individual functions. Finally, each optimization can be applied selectively by determining whether each potential application is among the code areas that justify improvement. We found it quite rewarding that relatively simplistic peephole optimization could improve performance this significantly.

These optimizations reveal that Java can be used to implement high-throughput servers and high-performance applications. For our server code, we are currently limited more by the throughput of our network hardware than the software processing. Indeed, as Table 9 indicates, InVerse can process packets faster than the network card can deliver and send data, even with a C++ receive loop. Ultimately, therefore, by applying the optimizations, we have effectively eliminated any negative impact of Java.

	Before Optimize	After Optimize
InVerse packets/sec. processed:		
1 User	145	810
10 Users	99	505
100 Users	N/A	147
Hardware limits (Java program)		
1 User		930
10 Users		93
100 Users		9
Hardware limits (C program)		
1 User		1080
10 Users		108
100 Users		11

Table 9: Impact of Java Optimizations on InVerse Performance (Packets/Second Processing Throughput) and Comparison to Hardware Capabilities

Unfortunately, we cannot guarantee that our current performance optimizations will retain their significance over time. Indeed, future releases of the Java virtual machine will have different performance characteristics, and improvements in JIT compilers are likely to mask many of the performance problems that we have observed. Furthermore, we are not certain that the characteristics exhibited by Sun's class libraries and virtual machine are shared by other Java implementations. We intend to execute our performance tests on other Java platforms. Furthermore, as technology advances, we will need to re-evaluate our optimizations to determine their continued value.

We are also continuing to study the performance of Java to find ways for improving application performance. We are beginning to look at how function locality within class files may impact memory layout within the virtual machine and the consequent speed of program and data access. We are also experimenting with how identifier name lengths may impact the caching behavior of byte code instruction access, and we are looking at additional techniques for improving performance by isolating common code paths from exception code. Finally, we are exploring the feasibility of incorporating these techniques into a performance analysis tool that might statically analyze application source code and determine opportunities for improving performance.

Java is widely regarded as the enabling technology for delivering dynamic content to users over the Internet. As network computing becomes ubiquitous, we foresee considerable advantages in employing a common implementation platform for the content and for the servers that deliver that content. The potential

advantages of Java-aware servers---including creation and delivery of Java objects to clients, elimination of cross-language calling constructs and Object Request Brokers, and integration with data stored on heterogeneous platforms---are exciting. Our performance analyses indicate that this single language environment can be a reality.

Acknowledgements

We would like to thank James Clinton for providing feedback on an earlier draft of this paper.

References

- [AG96] K. Arnold and J. Gosling. *The Java Programming Language* (Reading, Massachusetts: Addison Wesley), 1996.
- [C96] D. Cheriton. *Object-Oriented Programming from a Modeling and Simulation Perspective*. To appear.
- [d96] Wim dePaul. Personal Communication. November 1996.
- [HW96] J. Hartman and J. Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web* (Reading, Massachusetts: Addison-Wesley), 1996.
- [LY97] T. Lindholm and F. Yellin. *The JavaTM Virtual Machine Specification* (Reading, Massachusetts: Addison-Wesley), 1997.
- [S+97] S. Singhal, B. Nguyen, R. Redpath, J. Nguyen, and M. Fraenkel. "InVerse: Designing an Interactive Universe Architecture for Scalability and Extensibility." Submitted for publication, February 1997.
- [VH95] Arthur Van Hoff. Personal Communication. December 1995.
- [VRML96] VRML Architecture Group. "Living Worlds: Specification and Comments." October 1996.

Appendix: Performance Test Code

Thread Contention Source Code

```
import java.awt.*;

class thread0 extends Thread{
    long count =0;
    long t;
    final long MAXCOUNT= 1000000;

    public final void run(){
        t=System.currentTimeMillis();
        while (true) {
            TestSync.unsyncdummy();
//Test call
            if (++count>MAXCOUNT)
                break;
        }

        System.out.println(System.currentTimeMillis()-t
        );
    }
}

class thread1 extends Thread{
    long count =0;
    long t;
    final long MAXCOUNT= 1000000;

    public final void run(){
        t=System.currentTimeMillis();
        while (true) {
            TestSync.unsyncdummy();
//Test call
            if (++count>MAXCOUNT)
                break;
        }

        System.out.println(System.currentTimeMillis()-t
        );
    }
}

public class TestSync extends Frame {
    static public void unsyncdummy() {
    }
    static public synchronized void syncdummy()
    {
    }

    public static void main(String [] args){
        TestSync test = new TestSync();
        thread0 t0 = new thread0();
        thread1 t1 = new thread1();
```

```
t0.start();
t1.start();

        test.resize(400,100);
        test.show();
    }
}
```

No Thread Contention Source Code

```
public class TestSyncNoThread {

    static public void unsyncdummy() {
    }

    static public synchronized void syncdummy()
    {
    }

    public static void main(String [] args){
        int MAXCOUNT = 1000000;
        long t=System.currentTimeMillis();

        for (long i=0; i<MAXCOUNT; i++)
            TestSyncNoThread.unsyncdummy();
//This line modified

        System.out.println(System.currentTimeMillis()-t
        );
    }
}
```

Sync DataOutputStream Source Code

```
import java.io.*;

public class TestDataOutput {

    public static void main(String [] args){
        long MAXCOUNT = 50000;
        CopyByteArrayOutputStream bytea = new
        CopyByteArrayOutputStream(8*MAXCOUNT);
        CopyDataOutputStream dataao = new
        CopyDataOutputStream(bytea);
        long t=System.currentTimeMillis();
// try {
        for (long i=0; i<MAXCOUNT; i++) {
// dataao.writeByte(0);
// dataao.writeShort(0);
// dataao.writeChar(0);
// dataao.writeInt(0);
// dataao.writeLong(0);
```

Last revised 2/5/97

```
//      datao.writeFloat(0);
//      datao.writeDouble(0);
//    }
//    } catch (IOException io) {
//      System.out.println("IO exeception ");
//    }

System.out.println(System.currentTimeMillis()-t);
}

l = System.currentTimeMillis();
for (int j=0; j<count; j++) {
  temp=0xFFFFFFFF;
  for (int i=0; i<8; i++) {
    if ((temp&0x01)==0x01) {}
    if ((temp&0x02)==0x02) {}
    if ((temp&0x04)==0x04) {}
    if ((temp&0x08)==0x08) {}
    temp>>=4;
  }
}
```

Shift Source Code

```
public static void main(String [] args){
  long temp=0;
  long l;
  long count=50000;

  l = System.currentTimeMillis();
  for (int j=0; j<count; j++) {
    temp=0xFFFFFFFF;
    for (int i=0; i<32; i++) {
      }
    }

System.out.println(System.currentTimeMillis()-l);

  l = System.currentTimeMillis();
  for (int j=0; j<count; j++) {
    temp=0xFFFFFFFF;
    for (int i=0; i<32; i++) {
      if ((temp&0x01)==0x01) {}
      temp>>=1;
    }
  }

System.out.println(System.currentTimeMillis()-l);

  l = System.currentTimeMillis();
  for (int j=0; j<count; j++) {
    temp=0xFFFFFFFF;
    for (int i=0; i<16; i++) {
      if ((temp&0x01)==0x01) {}
      if ((temp&0x02)==0x02) {}
      temp>>=2;
    }
  }

System.out.println(System.currentTimeMillis()-l);
}
```

```
System.out.println(System.currentTimeMillis()-l);

l = System.currentTimeMillis();
for (int j=0; j<count; j++) {
  temp=0xFFFFFFFF;
  for (int i=0; i<4; i++) {
    if ((temp&0x01)==0x01) {}
    if ((temp&0x02)==0x02) {}
    if ((temp&0x04)==0x04) {}
    if ((temp&0x08)==0x08) {}
    if ((temp&0x01)==0x10) {}
    if ((temp&0x02)==0x20) {}
    if ((temp&0x04)==0x40) {}
    if ((temp&0x08)==0x80) {}
    temp>>=8;
  }
}

System.out.println(System.currentTimeMillis()-l);

l = System.currentTimeMillis();
for (int j=0; j<count; j++) {
  temp=0xFFFFFFFF;
  for (int i=0; i<2; i++) {
    if ((temp&0x01)==0x01) {}
    if ((temp&0x02)==0x02) {}
    if ((temp&0x04)==0x04) {}
    if ((temp&0x08)==0x08) {}
    if ((temp&0x01)==0x10) {}
    if ((temp&0x02)==0x20) {}
    if ((temp&0x04)==0x40) {}
    if ((temp&0x08)==0x80) {}

    if ((temp&0x01)==0x0100) {}
    if ((temp&0x02)==0x0200) {}
    if ((temp&0x04)==0x0400) {}
    if ((temp&0x08)==0x0800) {}
    if ((temp&0x01)==0x1000) {}
    if ((temp&0x02)==0x2000) {}
    if ((temp&0x04)==0x4000) {}
    if ((temp&0x08)==0x8000) {}
  }
}
```

```

        temp>>=16;
    }
}

System.out.println(System.currentTimeMillis()-l
);

l = System.currentTimeMillis();
for (int j=0; j<count; j++) {
    temp=0xFFFFFFFF;
    if ((temp&0x01)==0x01) {}
    if ((temp&0x02)==0x02) {}
    if ((temp&0x04)==0x04) {}
    if ((temp&0x08)==0x08) {}
    if ((temp&0x01)==0x10) {}
    if ((temp&0x02)==0x20) {}
    if ((temp&0x04)==0x40) {}
    if ((temp&0x08)==0x80) {}

    if ((temp&0x01)==0x0100) {}
    if ((temp&0x02)==0x0200) {}
    if ((temp&0x04)==0x0400) {}
    if ((temp&0x08)==0x0800) {}
    if ((temp&0x01)==0x1000) {}
    if ((temp&0x02)==0x2000) {}
    if ((temp&0x04)==0x4000) {}
    if ((temp&0x08)==0x8000) {}

    if ((temp&0x01)==0x00010000) {}
    if ((temp&0x02)==0x00020000) {}
    if ((temp&0x04)==0x00040000) {}
    if ((temp&0x08)==0x00080000) {}
    if ((temp&0x01)==0x00100000) {}
    if ((temp&0x02)==0x00200000) {}
    if ((temp&0x04)==0x00400000) {}
    if ((temp&0x08)==0x00800000) {}

    if ((temp&0x01)==0x01000000) {}
    if ((temp&0x02)==0x02000000) {}
    if ((temp&0x04)==0x04000000) {}
    if ((temp&0x08)==0x08000000) {}
    if ((temp&0x01)==0x10000000) {}
    if ((temp&0x02)==0x20000000) {}
    if ((temp&0x04)==0x40000000) {}
    if ((temp&0x08)==0x80000000) {}
}

System.out.println(System.currentTimeMillis()-l
);
}
}

```

Variable Access Source Code

```

class warm {
    public int warmvar=1;
}

class hot extends warm {
    public int hotvar=2;
}

class base extends hot {
    public int basevar=3;
}

public class TestVar {
    static public int    staticvar=200;

    public static void main(String [] args){
        int MAXCOUNT = 2000000;
        int local = 300;
        long l;

        warm clswarm = new warm();
        hot  clshot  = new hot();
        base clsbase = new base();

        l = System.currentTimeMillis();
        for (long i=0; i<MAXCOUNT; i++) {
            //      local = 400;
        }

        System.out.println(System.currentTimeMillis()-l
);

        l = System.currentTimeMillis();
        for (long i=0; i<MAXCOUNT; i++) {
            //      clsbase.basevar = 7;
        }

        System.out.println(System.currentTimeMillis()-l
);

        l = System.currentTimeMillis();
        for (long i=0; i<MAXCOUNT; i++) {
            //      clshot.hotvar = 7;
        }

        System.out.println(System.currentTimeMillis()-l
);

        l = System.currentTimeMillis();
        for (long i=0; i<MAXCOUNT; i++) {
            //      clswarm.warmvar = 7;
        }
    }
}

```

Last revised 2/5/97

```
    }

    System.out.println(System.currentTimeMillis()-l
);

    l = System.currentTimeMillis();
    for (long i=0; i<MAXCOUNT; i++) {
//        static var= 7;
    }

    System.out.println(System.currentTimeMillis()-l
);
}

System.out.println(System.currentTimeMillis()-t
);
try {
    for (long i=0; i<MAXCOUNT; i++) {
//        sb.writeByte(0);
//        sb.writeShort(0);
//        sb.writeChar(0);
//        sb.writeInt(0);
//        sb.writeLong(0);
//        sb.writeFloat(0);
        sb.writeDouble(0);
    }
} catch (IOException io) {
    System.out.println("IO exeception ");
}
```

currentTimeMillis() Source Code

```
import java.io.*;
import
ibm.inverse.util.ByteArrayDataOutputStream;
import

ibm.inverse.util.SynchByteArrayDataOutputStream;

public class TestIO {

    public static void main(String [] args){
        int MAXCOUNT = 50000;
        byte data0[] = new byte[MAXCOUNT*8];
        byte data1[] = new byte[MAXCOUNT*8];
        long t;
        ByteArrayDataOutputStream ba = new
        ByteArrayDataOutputStream(data0);
        SynchByteArrayDataOutputStream sb =
        new SynchByteArrayDataOutputStream(data1);

        t=System.currentTimeMillis();
        try {
            for (long i=0; i<MAXCOUNT; i++) {
//                ba.writeByte(0);
//                ba.writeShort(0);
//                ba.writeChar(0);
//                ba.writeInt(0);
//                ba.writeLong(0);
//                ba.writeFloat(0);
                ba.writeDouble(0);
            }
        } catch (IOException io) {
            System.out.println("IO exeception ");
        }
    }
}
```

```
System.out.println(System.currentTimeMillis()-t
);
    }
}
```

Try/Catch Source Code

```
public class TestTry {
    static public Exception cache= new Exception
();

    static public void functionCall() {
    }

    static public void tfunctionCall() throws
Exception {
        throw cache;
    }

    public static void main(String [] args){
        int MAXCOUNT =1000000;
        long l;

        l = System.currentTimeMillis();
        for (long i=0; i<MAXCOUNT; i++) {
            functionCall();
        }

        System.out.println(System.currentTimeMillis()-l
);

        l = System.currentTimeMillis();
        for (long i=0; i<MAXCOUNT; i++) {
            try {
                functionCall();
            } catch (Exception e) {
            }
        }
    }
}
```

Last revised 2/5/97

```
    }

    System.out.println(System.currentTimeMillis()-l
);

    l = System.currentTimeMillis();
    for (long i=0; i<MAXCOUNT; i++) {
        try {
            tfunctionCall();
        } catch (Exception e) {
        }
    }

    System.out.println(System.currentTimeMillis()-l
);

    }
}
```

Array bounds Source Code

```
public class TestArray {

    public static void main(String [] args){
        long MAXCOUNT =1000000;
        int array[] = new int [50];
        long l,j;
        int x=0;

        l = System.currentTimeMillis();
        for (int i=0; i <MAXCOUNT; i++) {
            j=0;
        }

        System.out.println(System.currentTimeMillis()-l
);

        /****
        l = System.currentTimeMillis();
        for (int i=0; i <MAXCOUNT; i++)
            if (i < array.length) {
                j=0;
            }

        System.out.println(System.currentTimeMillis()-l
);

        l = System.currentTimeMillis();
        for (int i=0; i <MAXCOUNT; i++)
            if ( i>0 && (i < array.length)) {
                j=0;
            }
        }
```

```
    }

    System.out.println(System.currentTimeMillis()-l
);

    l = System.currentTimeMillis();
    for (int i=0; i <MAXCOUNT; i++)
        array[x]=0;

    System.out.println(System.currentTimeMillis()-l
);

    l = System.currentTimeMillis();
    for (int i=0; i <MAXCOUNT; i++) {
        j=0;
        try {
            array[100]=0;
        } catch (ArrayIndexOutOfBoundsException
e) {
        }
    }

    System.out.println(System.currentTimeMillis()-l
);
    *****/
    }
}
```

HashLink/Hashtable Source Code

```
import ibm.inverse.util.HashLink;
import ibm.inverse.util.HashEntry;
import java.util.*;

public class hash {

    // Test function
    public static void main(String[] args){

        int loop = 50000;
        String[] array = new String[loop];
        for (int i = 0; i<loop; i++)
            array[i] = String.valueOf(i);

        HashLink link = new HashLink();
        Hashtable table = new Hashtable();

        Runtime.getRuntime().gc();
    }
}
```


Last revised 2/5/97

```
    long hashlinklookup =
System.currentTimeMillis();
    for (int i=0; i<loop; i++) {
        link.get(array[i]);
    }
    hashlinklookup =
System.currentTimeMillis()-hashlinklookup;

    long hashlookup = System.currentTimeMillis();
    for (int i=0; i<loop; i++) {
        table.get(array[i]);
    }
    hashlookup =
System.currentTimeMillis()-hashlookup;

    long t1 = System.currentTimeMillis();
    for (int i=0; i<loop; i++) {
        link.put(array[i], array[i]);
    }
    long t2 = System.currentTimeMillis();

    long t3 = System.currentTimeMillis();
    for (int i=0; i<loop; i++) {
        table.put(array[i], array[i]);
    }

    long sand = System.currentTimeMillis();
    for (int i=0; i<loop; i++) {
        link.get(array[i]);
    }
    sand = System.currentTimeMillis()-sand;
    System.out.println("Link gets on full hashlink
"+sand);

    sand = System.currentTimeMillis();
    for (int i=0; i<loop; i++) {
        table.get(array[i]);
    }
    sand = System.currentTimeMillis()-sand;
    System.out.println("Link gets on full hashtable
"+sand);

    long t4 = System.currentTimeMillis();

    System.out.println("Storing "+loop+" strings");
    System.out.println("  HashLink = "+(t2-t1)+" ...
Hashtable = "+(t4-t3));

    String s;
    t1 = System.currentTimeMillis();
    for (HashEntry e = link.getLinkedList(); e!= null;
e = e.next) {
        s = (String)e.value;
        s = null;
    }
```

```
    }
    t2 = System.currentTimeMillis();
    t3 = System.currentTimeMillis();
    for (Enumeration e = table.elements();
e.hasMoreElements();){
        s = (String)e.nextElement();
        s = null;
    }
    t4 = System.currentTimeMillis();
    System.out.println("Iteration "+loop+"
elements");
    System.out.println("  HashLink = "+(t2-t1)+" ...
Hashtable = "+(t4-t3));

    t1 = System.currentTimeMillis();
    for (int i = loop; i-- > 0;) {
        link.remove(array[i]);
    }
    t2 = System.currentTimeMillis();

    t3 = System.currentTimeMillis();
    for (int i = 0; i < loop; i++) {
        table.remove(array[i]);
    }
    t4 = System.currentTimeMillis();
    System.out.println("Removing "+loop+"
elements");
    System.out.println("  HashLink = "+(t2-t1)+" ...
Hashtable = "+(t4-t3));

    System.out.println("lookup "+loop+" strings");
    System.out.println("  HashLink =
"+hashlinklookup+" ...  Hashtable = "+hashlookup);
    }
}
```

Variable Read Access Source Code

```
class warm {
    public int warmvar=1;
}

class hot extends warm {
    public int hotvar=2;
}

class base extends hot {
    public int basevar=3;
}

public class TestRead {
    static public int  staticvar=200;

    public static void main(String [] args){
        int MAXCOUNT = 2000000;
        int local = 300;
```

Last revised 2/5/97

```
long l;  
int x;
```

```
warm clswarm = new warm();  
hot clshot = new hot();  
base clsbase = new base();
```

```
l = System.currentTimeMillis();  
for (long i=0; i<MAXCOUNT; i++) {  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
l = System.currentTimeMillis();  
for (long i=0; i<MAXCOUNT; i++) {  
    x = 7;  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
l = System.currentTimeMillis();  
for (long i=0; i<MAXCOUNT; i++) {  
    x=local;  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
l = System.currentTimeMillis();  
for (long i=0; i<MAXCOUNT; i++) {  
    x=clsbase.basevar;  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
l = System.currentTimeMillis();  
for (long i=0; i<MAXCOUNT; i++) {  
    x=clshot.hotvar;  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
l = System.currentTimeMillis();
```

```
for (long i=0; i<MAXCOUNT; i++) {  
    x=clswarm.warmvar;  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
l = System.currentTimeMillis();  
for (long i=0; i<MAXCOUNT; i++) {  
    x=staticvar= 7;  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);  
}
```

New method Source Code

```
class myobject {  
}
```

```
class cmyobject {  
    public cmyobject() {  
    }  
}
```

```
class mysubclass extends myobject {  
}
```

```
class cmysubclass extends cmyobject {  
}
```

```
public class TestNew {  
  
    public static void main(String [] args){  
        int MAXCOUNT = 500000;  
        long l;  
        myobject my;  
        cmyobject cmy;  
        mysubclass mysub;  
        cmysubclass cmysub;  
  
        l = System.currentTimeMillis();  
        for (long i=0; i<MAXCOUNT; i++) {  
        }  
    }  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
l = System.currentTimeMillis();
```

Last revised 2/5/97

<pre>for (long i=0; i<MAXCOUNT; i++) { my= new myobject(); } System.out.println(System.currentTimeMillis()-l); l = System.currentTimeMillis(); for (long i=0; i<MAXCOUNT; i++) { cmy= new cmyobject(); } System.out.println(System.currentTimeMillis()-l); l = System.currentTimeMillis(); for (long i=0; i<MAXCOUNT; i++) { mysub= new mysubclass(); } System.out.println(System.currentTimeMillis()-l); l = System.currentTimeMillis(); for (long i=0; i<MAXCOUNT; i++) { cmysub= new cmysubclass(); } System.out.println(System.currentTimeMillis()-l);</pre>	<pre>} System.out.println(System.currentTimeMillis()-l); l = System.currentTimeMillis(); for (int i=0; i<limit; i++){ p = (Panel) obj; } System.out.println(System.currentTimeMillis()-l); l = System.currentTimeMillis(); for (int i=0; i<limit; i++){ try { m = (Menu) obj; } catch (Exception e){ } } System.out.println(System.currentTimeMillis()-l); }</pre>
<hr/>	
Object creation/Array Access Source Code	
<pre>import java.util.*;</pre> <p> </p> <pre>public class Test {</pre>	<pre>}</pre>

Type cast Source Code

```
import java.awt.*;

public class instance {

    public static void main(String [] args){
        int limit =100000;
        long l;
        Object obj = (Object) new Panel();
        Panel p;
        Menu m;

        l = System.currentTimeMillis();
        for (int i=0; i<limit; i++)
            if (obj instanceof Panel) {
                p = (Panel) obj;
```

Object creation/Array Access Source Code

```
import java.util.*;

public class Test {

    Test() {}

    public static void main(String [] args){
        int MAXCOUNT = 1000000;
        Test [] array = new Test[MAXCOUNT];
        long l;
        array[0] = new Test();

        for (int i=1; i<MAXCOUNT; i++)
            array[i] = array[0];

        l = System.currentTimeMillis();
        for (int i=0; i<MAXCOUNT; i++) {
        }

        System.out.println(System.currentTimeMillis()-l);
    }
}
```

Last revised 2/5/97

```
Test x;
```

```
l = System.currentTimeMillis();  
for (int i=0; i<MAXCOUNT; i++) {  
    x = new Test();  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);
```

```
int count = MAXCOUNT-1;
```

```
l = System.currentTimeMillis();  
for (int i=0; i<MAXCOUNT; i++) {  
    synchronized(array){  
        x = array[count--];  
    }  
}
```

```
System.out.println(System.currentTimeMillis()-l  
);  
}
```