Source: https://www.geeksforgeeks.org/12-tips-to-optimize-java-code-performance/

12 Tips to Optimize Java Code Performance

- Difficulty Level : Medium
- Last Updated : 02 Aug, 2022

While working on any Java application we come across the concept of optimization. It is necessary that the code which we are writing is not only clean, and without defects but also optimized i.e. the time taken by the code to execute should be within intended limits. In order to achieve this, we need to refer to the Java coding standards and review our code to make sure it is as per the standards.

But sometimes we are not having the time to actually review the code due to deadline constraints. In such cases, we are providing some tips which a developer can keep in mind while doing coding of any requirement so that he/she needs to make minimal changes to code for fixing the performance during the testing phase or before moving the same to production.

1. Avoid Writing Long Methods

The methods should not be too long and should be specific to perform single functionality. It is better for maintenance as well as performance since while class loading and during method call, the method is loaded in stack memory. If methods are large with too much processing they will consume memory as well as CPU cycles to execute. Try to break the methods into smaller ones at suitable logical points. Also, if you want to strengthen your skills in Java then enroll with Geeksforgeeks **Java Programming Foundation – Self-Paced course** and learn the basic concepts, data types, operators, functions & more.

2. Avoid Multiple If-else Statements

We use conditional statements in our code for decision-making. The conditional statements should not be overused. If we are using too many conditional if-else statements it will impact performance since **JVM** will have to compare the conditions. This can become worse if the same is used in looping statements like for, while, etc. If there are too many conditions in your business logic try to

group the conditions and get the boolean outcome and use it in the if statement. Also, we can think of using a switch statement instead of multiple if-else if possible. Switch statement has a performance advantage over if – else. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
if (condition1) {

    if (condition2) {

        if (condition3 || condition4) { execute ..}

        else { execute..}
```

Note: Above sample is to be avoided and use this as follows:

boolean result = (condition1 && condition2) && (condition3 || condition4)

3. Avoid Getting the Size of the Collection in the Loop

While iterating through any collection get the size of the collection beforehand and never get it during iteration. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
List<String> objList = getData();
for (int i = 0; i < objList.size(); i++) { execute code ..}
```

Note: Above sample is to be avoided and use this as follows:

```
List<String> objList = getData();
int size = objList.size();
for (int i = 0; i < size; i++) { execute code ..}
```

4. Avoid Using String Objects For Concatenation

A string is an immutable class the object created by String cannot be reused. So if we need to create a large string in case of SQL queries etc it is bad practice to concatenate the String object using the '+' operator. This will lead to multiple objects of String created leading to more usage of heap memory. In this case, we can use StringBuilder or StringBuffer, the former is preferential over the latter since it has a performance advantage due to non-synchronized methods. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
 String query = String1+String2+String3;
```

Note: Above sample is to be avoided and use this as follows:

StringBuilder strBuilder = new StringBuilder("");

strBuilder.append(String1).append(String2).append(String3);

String query = strBuilder.toString();

5. Use Primitive Types Wherever Possible

Usage of primitive types over objects is beneficial since the primitive type data is stored on stack memory and the objects are stored on heap memory. If possible, we can use primitive types instead of objects since data access from stack memory is faster than heap memory. So it is always beneficial to use int over Integer or double over Double.

6. Avoid Using BigDecimal Class

We know that BigDecimal class provides accurate precision for the decimal values. Over usage of this object hampers the performance drastically specifically when the same is used to calculate certain values in a loop. BigDecimal uses a lot of memory over long or double to perform calculations. If precision is not the constraint or if we are sure the range of the calculated value will not exceed long or double we can avoid using BigDecimal and use long or double with proper casting instead.

7. Avoid Creating Big Objects Often

There are certain classes that act as data holders within the application. These objects are heavy and their creation should be avoided multiple times. An example of such objects is the DB connection objects or system configuration objects or session objects for the user after login. These objects used a lot of resources while created. We should reuse these objects instead of creating them as creation will drastically hamper the application performance due to more memory usage. We should use the Singleton pattern wherever possible to create a single instance of the object and reuse it wherever required or clone the object instead of creating a new one.

8. Use Stored Procedures Instead of Queries

It is better to write stored procedures instead of complex and long queries and call them while processing. Stored procedures are stored as objects in the database and pre-compiled. The execution time of the stored procedure is less compared to the query with the same business logic as a query is compiled and executed every time wherever it is called through the application. Also, the stored procedure has an advantage in data transfer and network traffic since we are not transferring the complex query for execution every time to the database server.

9. Using PreparedStatement instead of Statement

While executing the SQL query through the application we use JDBC API and classes for the same. PreparedStatement has an advantage over Statement for parameterized query execution since the preparedstatement object is compiled once and executed multiple times. Statement object on other hand is compiled and executed every time it is called. Also**,** the prepared statement object is safe to avoid SQL injection attacks for Web Application security.

10. Use of Unnecessary Log Statements and Incorrect Log Levels

Logging is an integral part of any application and needs to be implemented efficiently in order to avoid performance hits due to incorrect logging and log levels. We should avoid logging big objects into code. Logging should be limited to specific parameters we need to monitor and not the whole

object. Also**,** the logging level should be kept to higher levels like DEBUG, ERROR, and not INFO. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
Logger.debug("User info : " + user.toString());
Logger.info("Method called for setting user data:" + user.getData());
```

Note: Above sample is to be avoided and use this as follows:

Logger.debug(“User info : ” + user.getName() + ” : login ID : ” + user.getLoginId());

Logger.info(“Method called for setting user data”);

11. Select Required Columns in a Query

While getting the data from the database we use select queries to get the data. Avoid selecting columns that are not necessary for further processing. Select only those columns which we will be required for further processing or display on the front end. Selecting too many columns causes a delay in query execution at the database end. Also, it increases network traffic from database to application which should be avoided. The sample is provided below as an illustration which is to be avoided as follows:

Illustration:

```
select * from users where user_id = 100;
```

Note: Above sample is to be avoided and use this as follows:

select user_name, user_age, user_gender, user_occupation, user_address from users where user_id = 100;

12. Fetch the Data Using Joins

While getting the data from multiple tables it is necessary to use the joins properly on tables. If the joins are not properly used or the tables are not normalized it will cause a delay in query execution leading to a performance hit for the application. Avoid using subqueries instead of joins as subqueries take more time for execution than joins. Create an index on columns of the table which are frequently used for improving the performance of query execution and reducing the latency of the application.

Source: https://medium.com/javarevisited/6-directions-for-java-performance-optimization-you-should-know-603384b96831

6 Directions for Java Performance Optimization You Should Know

Photo by Annie Spratt on Unsplash

In the last article "**Performance Optimization Theory**", I introduced various indicators of performance analysis, so that when doing performance optimization, there are specific optimization goals and measurement methods, and the optimization effect will not just stay in the intuitive sense. superior.

Performance Optimization Theory You Should Know as a Programmer

Many people have only a very shallow understanding of performance optimization because they usually only focus on…

levelup.gitconnected.com

Now that you know your optimization goals, what should you do next?

As programmers, in our daily work, we optimize the way, mainly technical means. This series of technical means can be roughly classified into 6 categories.

It can be seen that the optimization method focuses on the planning of computing resources and storage resources. There are many ways of exchanging space for time in optimization methods, but it is not advisable to only take care of calculation speed without considering complexity and space issues. What we need to do is to achieve the optimal state of resource utilization under the premise of taking care of performance.

1 Reuse optimization

When writing code, you will find that there is a lot of repetitive code that can be extracted and made into public methods. In this way, the next time you use it, you don't have to write it again.

This idea is reused. The above description is an optimization of coding logic, and for data access, there is the same multiplexing situation. Whether in life or coding, repetitive things happen all the time. If there is no reuse, work and life will be more tiring.

In software systems, when it comes to data multiplexing, the first thing that comes to our mind is buffering and caching. Pay attention to the difference between these two words, their meanings are completely different, many people are easily confused, I will briefly introduce them here.

Buffer is commonly used for the temporary storage of data and then batch transmission or writing. The sequential method is used to alleviate frequent and slow random writes between different devices. The buffer is mainly for writing operations.

Cache is commonly used for multiplexing of reading data, by caching them in a relatively high-speed area, the cache is mainly for reading operations.

In Java, database connection pools, thread pools, etc. are used very frequently. Since the cost of creating and destroying these objects is relatively high, we will temporarily store them after use. The next time we use them, we don't need to go through the time-consuming initialization operation again.

2 Computational optimization

2.1 Parallel execution

Today's `CPUs` are developing very fast, and most hardware is multi-core. To speed up the execution of a task, the fastest and optimal solution is to have it executed in parallel. There are the following three modes of parallel execution.

The first mode is multi-machine, which uses load balancing to split traffic or large calculations into multiple parts and process them at the same time. For example, `Hadoop` uses `MapReduce` to break up tasks and perform calculations on multiple machines at the same time.

The second mode is to use multiple processes. For example, `Nginx` adopting the `NIO` programming model. The Master manages the Worker process in a unified manner, and then the Worker process performs the real request proxy, which can also make good use of multiple `CPUs` of the hardware.

The third mode is to use multithreading, which is also the most exposed to Java programmers. For example, `Netty`, which uses the `Reactor` programming model, also uses `NIO`, but it is thread-based.

The Boss thread is used to receive the request and then dispatch it to the corresponding Worker thread for real business computation.

2.2 Change from synchronous to asynchronous

Synchronous to asynchronous, which usually involves a change in the programming model. In synchronous mode, the request will block until a success or failure result is returned. Although its programming model is simple, it is particularly problematic when dealing with sudden and skewed traffic, and requests can easily fail.

Asynchronous operations can easily support horizontal expansion, and can also relieve transient pressure and make requests smoother.

2.3 Lazy loading

The last one is to use some common design patterns to optimize business and improve the experience, such as **singleton patterns**, **proxy patterns**, etc. For example, when drawing a Swing window, if you want to display more pictures, you can load a placeholder first, and then slowly load the required resources through the background thread, which can avoid the window from being frozen.

3 Result set optimization

Next, we will introduce the optimization of the result set. To give a more intuitive example, we all know that the representation of `XML` is very good, so why is there `JSON`? In addition to being simpler to write, an important reason is that its volume has become smaller, and the transmission efficiency and parsing efficiency have become higher. Like `Google's` Protobuf, the volume is smaller. Although the readability is reduced, in some high concurrency scenarios (such as `RPC`), the efficiency can be significantly improved, which is a typical optimization of the result set.

Like `Nginx`, `GZIP` compression is generally turned on to keep the transmitted content compact. The client only needs a small amount of computing power, and it can be easily decompressed. Since this operation is decentralized, the performance penalty is fixed.

Understanding this truth, we can see the general idea of result set optimization, you should try to keep the returned data as concise as possible. Some fields that the client does not need, then in the code, or directly in the `SQL` query, remove it.

Some businesses that do not have high requirements for timeliness, but have high requirements for processing power. We need to learn from the experience of the buffer, minimize the interaction of network connections, and use batch processing to increase the processing speed.

The result set is likely to be used twice, and you may cache it, but it still lacks speed. At this time, it is necessary to optimize the processing of the data collection, using `indexes` or `Bitmap` bitmaps to speed up data access.

4 Resource conflict optimization

In our usual development, we will involve a lot of shared resources. Some of these shared resources are stand-alone, such as a `HashMap`; some are external storage, such as a database row; some are single resources, such as `setnx` of a key in `Redis`; some are coordination of multiple resources, such as transactions, distributed transactions, etc.

In reality, there are many performance issues related to locks. Most of us think of database row locks, table locks, various locks in `Java`, etc. At a lower level, such as `CPU` command-level locks, `JVM` instruction-level locks, operating system internal locks, etc., it can be said to be everywhere.

Only concurrency can generate resource conflicts. That is, at the same time, only one processing request can obtain the shared resource. The way to resolve resource conflicts is to lock. Another example is a transaction, which is essentially a lock.

According to the lock level, locks can be divided into optimistic locks and pessimistic locks. Optimistic locks are more efficient; according to lock types, locks are divided into fair locks and unfair locks. There are some subtleties in the scheduling of tasks. difference.

Competition for resources will cause serious performance problems, so there will be some research on lock-free queues, which will significantly improve performance.

5 Algorithm optimization

Algorithms can significantly improve the performance of the complex business, but in actual business, they are often variants. As storage is getting cheaper and cheaper, in some `CPU-intensive` businesses, space is often used for time to speed up processing.

Algorithms belong to code tuning. Code tuning involves a lot of coding skills and requires users to be very familiar with the `API` of the language used. Sometimes, the flexible use of algorithms and data structures is also an important part of code optimization. For example, commonly used ways to reduce time complexity include recursion, bisection, sorting, dynamic programming, etc.

A good implementation has a very large impact on the system than a poor implementation. For example, the implementation of `List`, `LinkedList` and `ArrayList` are several orders of magnitude worse in random access performance; another example `CopyOnWriteList` adopts the method of `copy-on-write`, which can significantly reduce the lock conflict in the scenario of reading more and writing less. When to use `synchronization` and when to be thread-safe, also have higher requirements on our coding ability.

In this part of the knowledge, we need to pay attention to accumulation in our everyday work.

6 JVM optimization

Because Java runs on a JVM virtual machine, many of its features are restricted by the JVM. Optimizing the JVM virtual machine can also improve the performance of JAVA programs to a certain extent. If the parameters are not configured properly, it may even cause serious consequences such as OOM.

JVM performance tuning involves various trade-offs, often affecting the whole body, and it is necessary to comprehensively consider the impact of all aspects. Therefore, it is very important to understand some of the internal operating principles of the JVM. It will help us deepen our understanding of the code and help us write more efficient code.

Finally

Thanks for reading. I am looking forward to your following and reading more high-quality articles.

omgzui

Source: https://betterprogramming.pub/6-techniques-for-java-performance-optimization-c6d853d24e6d

6 Techniques for Java Performance Optimization

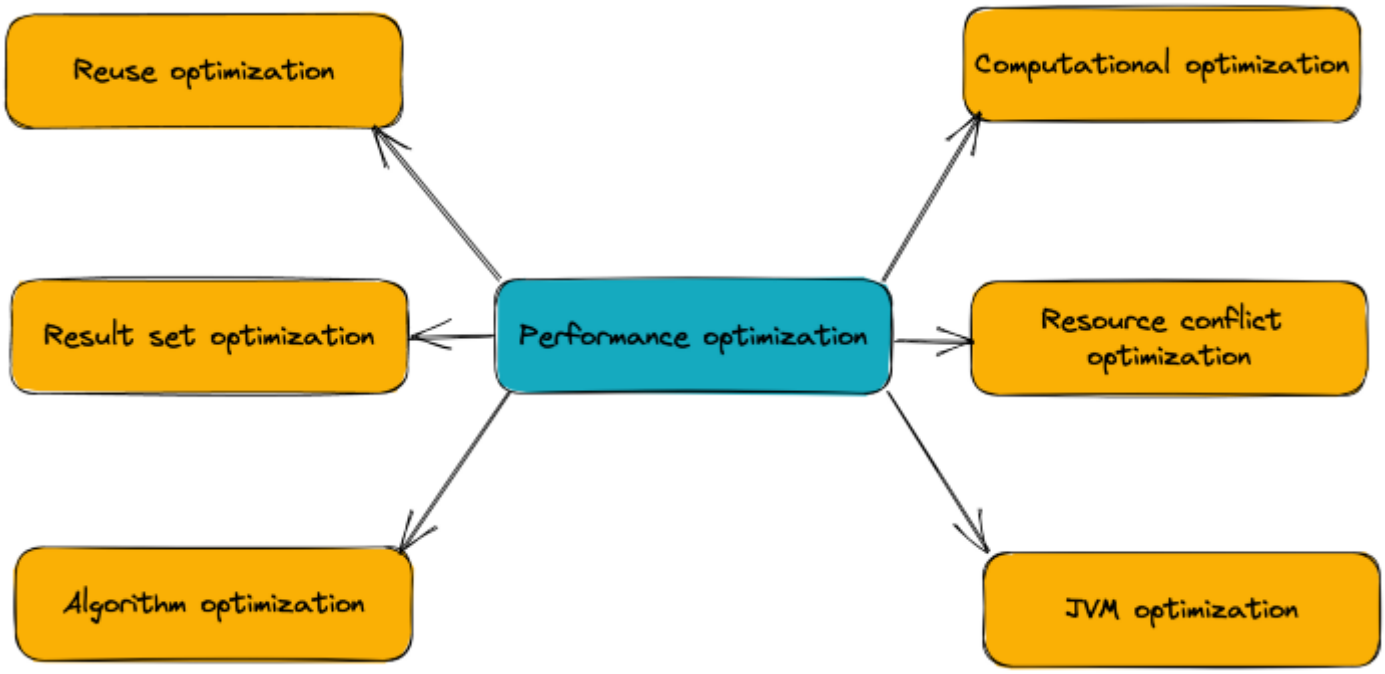
To speed up different areas of your Java projects

Photo by Christopher Gower on Unsplash

Performance optimization is divided into business optimization and technical optimization. The effects of business optimization are large, and they belong to the product and management category.

As developers, the optimization methods we face are mainly through a series of technical means to complete the established optimization goals in our daily work.

This series of technical means can be roughly classified into the following six categories:



Optimization focuses on planning computing resources and storage resources. There are many ways to exchange space for time in optimization methods, but it is not advisable to only take care of calculation speed without considering complexity and space issues.

We need to achieve optimal resource utilization to take care of performance. Here's how we can do that for each type of optimization:

Reuse Optimization

When writing code, you will find a lot of repetitive code can be extracted and made into public methods, so you don't have to write it again.

This idea is reuse. The above description is an optimization of coding logic, and for data access, there is the same multiplexing situation. Whether in life or in coding, repetitive things happen all the time. If there is no reuse, work and life will be more tiring.

When it comes to data multiplexing in software systems, the first thing that comes to our mind is buffering and caching. Note the difference between these two words: their meanings are completely different. Here is a brief introduction.

Buffer is commonly used to temporarily store data and then batch transmission or writing. The sequential method is used to alleviate frequent and slow random writes between different devices. The buffer is mainly for write operations.

Cache is commonly used for multiplexing of reading data by caching them in a relatively high-speed area. The cache is mainly for reading operations.

Similarly, object pooling operations, such as database connection pools, thread pools, etc., are used frequently in Java.

Since the cost of creating and destroying these objects is relatively high, we will temporarily store these objects after use, and the next time we use them, we don't need to go through the time-consuming initialization operation again.

Computational optimization

1. Parallel execution

Today's CPUs are developing rapidly, and most hardware is multi-core. To speed up the execution of a task, the fastest and optimal solution is to have it execute in parallel.

There are three modes of parallel execution:

The first mode is multimachine, which uses load balancing to split traffic or large calculations into multiple parts and process them at the same time. For example, Hadoop uses MapReduce to break up tasks and perform calculations on multiple machines at the same time.

The second mode uses multiple processes. For example, Nginx adopts the NIO programming model. The Master manages the Worker process in a unified manner, and then the Worker process performs the real request proxy, which makes good use of multiple CPUs of the hardware.

The third mode uses multithreading, which is also the most exposed to Java programmers. Netty, for example, uses the Reactor programming model and also uses NIO, but it is thread-based. The Boss thread is used to receive the request and then dispatch it to the corresponding Worker thread for real business calculations.

Languages like Golang have more lightweight coroutines. Coroutines are more lightweight than threads, but they are not mature in Java yet, so I won't introduce them too much. But in essence, it is also for multi-core applications, which allows tasks to be executed in parallel.

2. Synchronous to asynchronous

Another computing optimization changes modes from synchronous to asynchronous. Usually ,this involves a change in the programming model.

In synchronous mode, the request will block until a success or failure result is returned.

Although its programming model is simple, it is particularly problematic when dealing with sudden and skewed traffic, and requests can easily fail.

Asynchronous operations can easily support horizontal expansion and relieve transient pressure and make requests smoother.

Synchronous requests are like a fist on a steel plate; asynchronous requests are like a fist on a sponge.

You can imagine the process; the latter is elastic, and the experience is more friendly.

3. Lazy loading

The last one uses some common design patterns to optimize business and improve the experience, such as singleton patterns, proxy patterns, etc.

For example, if you want to display more pictures when drawing a Swing window, you can load a placeholder first and then slowly load the required resources through a background thread. This method can prevent window freezes.

Result set optimization

Next, we will introduce the optimization of the result set. For a more intuitive example, we all know that the representation of XML is very good, so why is there JSON?

In addition to being easier to write, an important reason is that its volume has become smaller, and the transmission efficiency and parsing efficiency have become higher. Like Google's Protobuf, the volume is smaller.

Although the readability is reduced, in some high concurrency scenarios (such as RPC), the efficiency can be significantly improved, which is typical of the result set.

This is because the current web services are in C/S mode. When data is transmitted from the server to the client, it needs to be distributed in multiple copies. This amount of data is rapidly expanding. Every time a small part of the storage is reduced, there will be a relatively large increase in transmission performance and cost.

Like Nginx, GZIP compression is generally turned on to keep the transmitted content compact. The client only needs a small amount of computing power, and it can be easily decompressed. Since this operation is decentralized, the performance penalty is fixed.

Understanding this truth, we can see the general idea of result set optimization. You should try to keep the returned data as concise as possible. If a client doesn't need some fields in the code or SQL query, remove them.

Some businesses do not have high timeliness requirements but have high processing power requirements. We need to learn from the experience of the buffer, minimize the interaction of network connections, and use batch processing to increase the processing speed.

The result set is likely to be used twice. You may cache it, but it still lacks speed. At this time, it is necessary to optimize the processing of the data collection using indexes or Bitmap bitmaps to speed up data access.

Resource conflict optimization

In our usual development, we will involve a lot of shared resources, such as the following:

- Standalone shared resources, such as a HashMap
- External storage, such as a database row.
- Single resources, such as `setnx` a key in Redis.
- Coordination of multiple resources, such as transactions, distributed transactions, etc.

In reality, there are many performance issues related to locks. Most of us think of database row locks, table locks, various locks in Java, etc, but these can be everywhere, especially at a lower level (i.e., CPU command-level locks, JVM instruction-level locks, operating system internal locks, etc.).

Only concurrency can generate resource conflicts, which means only one processing request can obtain the shared resource simultaneously.

The way to resolve resource conflicts is to lock. Another example is a transaction, which is essentially a lock.

According to the lock level, locks can be divided into optimistic and pessimistic locks. Optimistic locks are more efficient. According to lock types, locks are divided into fair locks and unfair locks. There are some subtleties in the scheduling of tasks' differences.

Competition for resources will cause serious performance problems, so there will be some research on lock-free queues, which will greatly improve performance.

Algorithm optimization

Algorithms can significantly improve the performance of complex business, but in actual business, they are often variants.

As storage gets cheaper in some CPU-intensive businesses, space is often used to speed up processing.

Algorithms belong to code tuning, which involves many coding skills and requires users to be very familiar with the API of the language used.

Sometimes, the flexible use of algorithms and data structures is also an important part of code optimization.

For example, commonly used ways to reduce time complexity include recursion, binary search, sorting, dynamic programming, etc.

An excellent implementation has a greater impact on the system than a poor implementation. For example, the implementation of `List`, `LinkedList`, and `ArrayList` are several orders of magnitude worse in random access performance

For example, `CopyOnWriteList` adopts the copy-on-write method, which can significantly reduce the lock conflict when need to read more and write less.

Knowing when to use synchronization and when to be thread-safe challenges our coding ability; it requires us to pay attention to accumulation in our ordinary work.

In regular programming, we try to use some components with good design concepts and superior performance. For example, with Netty, there is no need to choose the older Mina components.

When designing a system, considering performance factors, do not choose a time-consuming protocol such as SOAP.

Another example is a good parser (such as JavaCC); its efficiency will be much higher than regular expressions.

JVM optimization

Because Java runs on the JVM virtual machine, many of its features are restricted by the JVM.

Optimizing the JVM virtual machine can also improve the performance of JAVA programs to a certain extent.

If the parameter configuration is improper, it will even cause serious consequences such as OOM.

The currently widely used garbage collector is G1. With few parameter configurations, memory can be efficiently reclaimed.

The CMS garbage collector has been removed in Java 14. Due to its uncontrollable GC time, it should be avoided.

JVM performance tuning involves various trade-offs, often affecting the whole body, and it is necessary to consider the impact of all aspects.

Therefore, it is very important to understand some internal operating principles of the JVM. It will help us deepen our understanding of the code and help us write more efficient code.

Thank you for reading this article. If you find any errors in this article, please let me know.

Thanks to Anupam Chugh

Source: https://www.infoworld.com/article/2077594/java-tip-90--accelerate-your-guis.html

Java Tip 90: Accelerate your GUIs

Improve GUI performance using 'lazy' evaluation

Slow graphical user interfaces (GUIs) are a common complaint directed at Java. While constructing snappy GUIs can take time and effort, I'll present a class that can speed up GUIs with little extra work on your part. For serious GUI improvements, this is only the first step; however, it's an excellent one because it's simple. Sometimes this class improves performance enough without any additional work.

The approach

The approach is easily stated: delay building GUI components until necessary. On a macro scale, all Java programs already do this by default. Who, after all, builds all the possible frames and windows of a program while initializing? However, the class I present here allows you more fine-grained, lazy construction than that already offered by Java.

Three observations led to the utility class I built to support lazy construction of GUI components. First, most GUI panels are simple and initialize fast enough without any extra work.

Panels that cram a huge number of GUI components onto the screen are responsible for most slow initializations. These panels tend to be of two types, either tabbed panes or scroll panels.

Second, until a GUI component needs to be viewed, it usually doesn't need to be constructed. Third, GUI components cannot be seen until one of the following methods is called:

- `public void paint (Graphics)`
- `public void paintComponents(Graphics)`
- `public void paintAll (Graphics)`
- `public void repaint ()`
- `public void repaint (long)`
- `public void repaint (int, int, int, int)`
- `public void repaint (long, int, int, int, int)`
- `public void update (Graphics)`

The idea is to create a utility panel class that moves most of the GUI construction code out of the initializers and constructors. Instead, this code is placed in a function called `lazyConstructor()`. This function does most of the work ordinarily done by the GUI constructor, but is not itself a constructor. This utility panel class ensures that the `lazyConstructor` method is called once *before* any paint or update methods are called.

Using this panel class inside tab panels lets the tabbed panes construct faster because only one visible tab panel needs to be constructed initially.

The code

The code for this class looks like this:

```
import java.awt.*;

/**
 * LazyPanel is an abstract base class that provides functionality
 * to defer populating a Panel object until it is actually viewed.
 * This is extremely useful when using CardLayout and tab panel
 * views because it allows the construction of the subviews to
 * be done on a pay-as-you-go basis instead of absorbing all the cost
 * of construction up front.
 *
 * If subclasses choose to override any of the following methods,
 * it is their responsibility to ensure their overridden methods
 * call the parent's method
first
. The methods are:
 *
 * public void paint (Graphics)
 * public void paintComponents(Graphics)
 * public void paintAll (Graphics)
 * public void repaint ()
 * public void repaint (long)
 * public void repaint (int, int, int, int)
 * public void repaint (long, int, int, int, int)
 * public void update (Graphics)
 *
 * Each of these methods ensures the panel is constructed
 * and then simply forwards the call to the parent class.
 *
 * You use this class by extending it and moving as much of the
 * constructor code as possible from the child class into the method
 * lazyConstructor. An example of using LazyPanel is:
 *
*
 * <PRE>
 *     import java.awt.*;
 *
 *     class BusyPanel extends LazyPanel
 *     {
 *         public BusyPanel (int rows, int cols)
 *         {
 *             this.rows = rows;
 *             this.cols = cols;
 *         }
 *
 *         protected void lazyConstructor()
 *         {
 *             setLayout (new GridLayout (rows, cols));
 *             for (int i = 0; i < rows * cols; ++i)
 *             {
 *                 add (new Button (Integer.toString (i + startValue)));
 *                 ++startValue;
 *             }
 *         }
 *
 *         static private int startValue = 0;
```

```
 *
 *         private int rows;
 *         private int cols;
 *     }
 * </PRE>
 * You use it like this:
 * <PRE>
 *     import java.awt.*;
 *     import java.awt.event.*;
 *
 *     class TestFrame extends Frame
 *     {
 *         public TestFrame ()
 *         {
 *             setLayout (new BorderLayout ());
 *             add (nextButton, "South");
 *
 *             framePanel.setLayout (layout);
 *             for (int i = 0; i < 30; ++i)
 *                 framePanel.add (new BusyPanel (8, 8), "");
 *             add (framePanel, "Center");
 *
 *             nextButton.addActionListener (
 *                 new ActionListener()
 *                 {
 *                     public void actionPerformed (ActionEvent event)
 *                     {
 *                         layout.next (framePanel);
 *                     }
 *                 }
 *             );
 *
 *             setSize (400, 300);
 *         }
 *
 *         private CardLayout layout = new CardLayout();
 *         private Button nextButton = new Button ("Next Panel");
 *         private Panel  framePanel = new Panel();
 *
 *         static public void main (String args[])
 *         {
 *             (new TestFrame()).show();
 *         }
 *     }
 * </PRE>
 * To see the advantage of using the LazyPanel, try moving the code
 * in the lazyConstructor() method into the constructor of BusyPanel,
 * recompile and rerun the example. The extra lag in startup time
 * is what the LazyPanel class is intended to remove.
 *
 * This works with swing, too. Just modify the LazyPanel to
 * extend JPanel instead of Panel.
 */
public abstract class LazyPanel extends Panel
{
    // We want to call the lazyConstructor only once.
    private boolean lazyConstructorCalled = false;
    // Some versions of Swing called paint() before
    // the components were added to their containers.
    // We don't want to call lazyConstructor until
    // the components are actually visible.
    private boolean isConstructorFinished = false;
    /**
     * Make a LazyPanel.
```

```
     */
    protected LazyPanel ()
    {
        isConstructorFinished = true;
    }
    public void paint (Graphics g)
    {
        callLazyConstructor();
        super.paint (g);
    }
    public void paintAll(Graphics g)
    {
        callLazyConstructor();
        super.paintAll (g);
    }
    public void paintComponents (Graphics g)
    {
        callLazyConstructor();
        super.paintComponents (g);
    }
    public void repaint ()
    {
        callLazyConstructor();
        super.repaint();
    }
    public void repaint (long l)
    {
        callLazyConstructor();
        super.repaint (l);
    }
    public void repaint (int i1, int i2, int i3, int i4)
    {
        callLazyConstructor();
        super.repaint (i1, i2, i3, i4);
    }
    public void repaint (long l, int i1, int i2, int i3, int i4)
    {
        callLazyConstructor();
        super.repaint (l, i1, i2, i3, i4);
    }
    public void update (Graphics g)
    {
        callLazyConstructor();
        super.update (g);
    }
    /**
     * Force the lazyConstructor() method implemented in the child class
     * to be called. If this method is called more than once on
     * a given object, all calls but the first do nothing.
     */
    public synchronized final void callLazyConstructor()
    {
        // The general idea below is as follows:
        //     1) See if this method has already been successfully called.
        //        If so, return without doing anything.
        //
        //     2) Otherwise ... call the lazy constructor.
        //     3) Call validate so that any components added are visible.
        //     4) Note that we have run.

        if ((lazyConstructorCalled == false) && (getParent() != null))
        {
            lazyConstructor();
            lazyConstructorCalled = true;
```

```
            validate();
        }
    }
    /**
     * This method must be implemented by any child class. Most of
     * the component creation code that would have gone in the constructor
     * of the child goes here instead. See the
example

     * at the top.
     */
    abstract protected void lazyConstructor ();
}
```

Using the LazyPanel

Using the `LazyPanel` class is straightforward, but there are a few things of which you should be aware.

Don't throw exceptions from lazyConstructor

For panels, it is easy enough to write constructors that throw exceptions if something goes wrong. Unfortunately, `lazyConstructor` is an ordinary method and not a constructor. Because `lazyConstructor` is called from `paint()`, it cannot throw anything except `RuntimeExceptions` and `Errors`. Since there isn't any caller other than the AWT thread to catch these exceptions and errors, it is best not to throw any exceptions at all. If necessary, place the body of your `lazyConstructor` method in a try/catch block and do something sensible in the catch block.

GUI builders will not cooperate

The code generated by the GUI builders in Inprise's JBuilder and Symantec's Cafe does not work well with the `LazyPanel` class. The GUI builders' generated code uses initializers to construct most of the GUI components for the panel. This eliminates the advantage of the `LazyPanel`, since none of the work is moved to the `lazyConstructor()` method.

One approach is to manually move the component construction from the initializers to the `lazyConstructor` method. Unfortunately, the GUI builder then can no longer modify the panel for further enhancements. A better approach is to keep the GUI-builder-generated panel unmodified and put it inside a `LazyPanel`. The `LazyPanel` then contains only one component -- the component built by the GUI builder.

Your code will not perform any faster in the long run

Finally, this approach doesn't actually speed up your application. If your GUI originally needed 1 MB of memory and took 30 seconds to create all the components, it will still need 1 MB of memory

and will still take 30 seconds to fully construct. The `LazyPanel` spreads the cost of constructing the needed GUI components over time, so you pay the price in small chunks instead of requiring all the memory and time up front. `LazyPanel` does not, however, do anything to reduce that cost. Often this class will be good enough to accelerate your GUI. Other times, however, simply dropping in a `LazyPanel` or two will only begin your serious performance tuning.

Conclusion

Java GUIs with many components initialize more quickly by completing only the absolute minimum initialization necessary. The `LazyPanel` class lets you speed up the initialization of more complex GUIs with little extra programming. It is a good start for improving the speed of complex GUIs and often is sufficient for less complicated ones.

Mark Roulo is the Java Tip technical coordinator for JavaWorld. He has been programming professionally since 1989 and has been using Java since the alpha-3 release. He works full time at KLA-Tencor and is part of a team building a 500KLOC, distributed, parallel, multicomputer application for image processing (among other things) written almost entirely in Java.

Learn more about this topic

- Documentation on the `CardLayout`
  http://java.sun.com/products/jdk/1.2/docs/api/java/awt/CardLayout.html
- A tutorial on tabbed panes
  http://java.sun.com/docs/books/tutorial/uiswing/components/tabbedpane.html

This story, "Java Tip 90: Accelerate your GUIs" was originally published by JavaWorld.

Related:

- Build Automation
- Learn Java

Source: https://opensource.com/life/16/8/5-great-tricks-java-performance-optimization

5 great Java performance optimization tricks

By Dr. Michael J. Garbade
August 31, 2016 | 1 Comment | %t min read

Image by:

Photo by Jen Wike Huger

Optimizing your Java code requires an understanding of how the different elements in Java interact, and how it interacts with the operating system that is it running on. Use these five tips and resources to start learning how to analyze and optimize your code.

Before we get to the good stuff, you might be concerned about licensing. Java is owned by Oracle, and is under Oracle's BCL license, which is not a free/open source license. Even so, Oracle Java is part of many open source projects. OpenJDK is the free software implementation of the Java platform, licensed under GPL v2. (See Free Java implementations on Wikipedia for more information.)

Getting started

Performance optimization depends on multiple factors including garbage collection, virtual machines, and underlying operating system (OS) settings. There are multiple tools that a developer can use for analysis and optimization purposes, and you can learn about some of them by reading *Java Tools for Source Code Optimization and Analysis*. If you are struggling with terminology and Java fundamentals, check out the Livecoding Java category page for livestreams, archived videos, and other useful information.

"It depends"

It is necessary to understand that no two applications can have the same optimizations, and that there is no sure-shot path to optimize Java applications. It is all about using the best practices and sticking to a proper way of handling performance optimization methodology. To be really at the top of performance optimization you, as a Java developer, need to have proper understanding of the Java Virtual Machine (JVM) and the underlying operating system:

- **JVM and underlying OS:** The Java Virtual Machine is the home of any Java application. Read the JVM internals guide to learn more about JVM internals, and operating system differences.
- **JVM Distribution Model:** The Java Distribution Model deals with multiple JVM instances for your application. The distribution model improves app performance as it gets more resources to work with. You can go forward with two approaches. The first approach is to run multiple JVMs on a single server with a heap size of either 2GB or 8GB. The second approach is to run one JVM on multiple servers. Choosing the right approach depends on multiple factors including availability and responsiveness.
- **JVM Architecture:** Choosing the right JVM architecture is important for performance. You can go with either a 64-bit JVM machine or a 32-bit JVM machine. Generally, the 32-bit JVM performs better than its 64-bit counterpart. The only reason you should choose the 64-bit JVM is if you require a heap size greater than 3GB.

With the basic ideas of performance optimization and its elements clear, we will now focus on tricks that help you optimize your Java application.

1. Tune garbage collection (GC)

It is very hard to find the exact performance of your application due to the garbage collection complexity. However, if you really want to optimize your application, you need to handle garbage collection accordingly. The general rule is to change GC settings and perform profiling at the same time.

Once you are satisfied with the results, you can stop the process and move towards other forms of optimization. Ensure that in addition to average transaction time you are also looking out for outliers. The outliers are the real culprits when it comes to the slowing down of the Java application, and are hard to find.

Additionally, you need to understand the impact of performance hits during application runtime. A slowdown every single week can be ignored, whereas a slowdown every single database transaction can be a costly affair. Choose your optimization path accordingly, and optimize the application according to the workload.

2. Get the right GC algorithm to work for you

Let's get more into the GC optimization. After all, it is the meat of the whole optimization problem at hand. Currently, there are four Java garbage collector algorithms from which you can choose. Each of the algorithms cater to different needs, so you need to choose accordingly. Many developers fail to optimize their applications because they have no idea about GC algorithms.

The four algorithms are the Serial Collector, Parallel/Throughput collector, CMS collector, and the G1 collector. To learn more about each of the garbage collectors and how they work, check out the amazing *Garbage Collectors—Serial vs. Parallel vs. CMS vs. G1* from the Takipi blog. The article also discusses the Java 8 impact on the GC algorithm and other minute changes.

Coming back to the GC algorithm, according to *Understanding Java Garbage Collection*, the Concurrent Mark & Sweep GC (or "CMS") algorithm is the best algorithm choice for web server applications. Parallel GC algorithm is great for the application that has built-in predictability.

G1 and CMS are ideal for concurrent operations, but will also cause frequent pauses. The choice also depends on the trade-offs. For example, it is a good idea to choose parallel algorithm even when it has longer GC pause time compared to other GC algorithms.

3. Java heap

The Java memory heap plays a crucial role in keeping up with the memory requirement. It is always better to start with minimum heap allocation, then increase it with continued testing. Most of the time the optimization problem is solved by increasing heap size, but the solution doesn't work if there is a lot of GC overhead.

GC overhead also makes throughput too low, making the application undesirably slow. Furthermore, tuning GC earlier can help you avert the problems with heap size allocation. To get started, you can choose any heap size from 1GB to 8GB. The concept of old generation and new generation objects also kicks in while choosing the right heap size.

In the end, the heap size should depend on an old generation to new generation object ratio, previous GC optimization, and liveset, which is the memory size of objects.

4. Core app optimization

Core code optimization is the best way to optimize your Java application. If your application is not responding to GC and heap optimization, it is better to do architectural changes and concentrate on how your application processes information. Using clever algorithms and taking care of objects can solve a lot of problems including fragmentation, heap issues, and garbage collection issues.

5. Using optimal functions

Java has multiple functions to handle algorithmic performance. If you use StringBuilder instead of simple String, you will gain few improvements in performance. However, there are other ways to handle optimization at the code level. Let's look at them below.

- Use StringBuilder instead of the + operator.
- Avoid using the iterator().
- Take maximum benefit of the stack.
- Avoid regular expressions and instead use Apache Commons Lang.
- Stay away from recursion. Recursions are very resource intensive!

Read more about Java code optimization in *Top 10 Easy Performance Optimisations in Java*.

Conclusion

Java performance optimization is a big subject, and this article clearly doesn't cover everything. If you think something needs to be added to the article, don't forget to share it with the audience by commenting below.

Tags
Java
Dr. Michael J. Garbade

Dr. Michael is the founder and CEO of Los Angeles-based Education Ecosystem. Inc. (previously Livecoding.tv). Education Ecosystem (LEDU) is a project-based learning platform that teaches students how to build real products in areas such as programming, game development, artificial intelligence, cybersecurity, data science, and blockchain.

Source: https://www.javaperformancetuning.com/tips/rawtips.shtml

Java Performance Tuning

Java(TM) - see bottom of page

Java performance tuning tips

Note that this page is very large. The tips on this page are categorized in other pages. Use the tips index page to access smaller focused listings of tips.

This page lists many other pages available on the web, together with a condensed list of tuning tips that each page includes. For the most part I've eliminated any tips that are wrong, but one or two may have slipped past me. Remember that **the tuning tips listed are not necessarily good coding practice. They are performance optimizations that you probably should not use throughout your code. Instead they apply to speeding up critical sections of code where performance has already been identified as a problem.**

The tips here include only those that are available online for free. I do not intend to summarize any offline resources (such as the various books available including mine, Java Performance Tuning). The tips here are of very variable quality and usefulness, some real gems but some dross and quite a bit of repetition. Comments in square brackets, [], have been added by me.

Use this page by using your browser's "find" or "search" option to identify particular tips you are interested in on the page, and follow up by reading the referenced web page if clarification is necessary.

This page is currently 411KB. This page is updated once a month. You can receive email notification of any changes by subscribing to the newsletter

---

http://www.onjava.com/pub/a/onjava/2001/02/22/optimization.html
Performance planning for managers (Page last updated February 2001, Added 2001-03-21, Author Jack Shirazi, Publisher OnJava). Tips:

- Include budget for performance management.
- Create internal performance experts.
- Set performance requirements in the specifications.
- Include a performance focus in the analysis.
- Require performance predictions from the design.
- Create a performance test environment.
- Test a simulation or skeleton system for validation.
- Integrate performance logging into the application layer boundaries.
- Performance test the system at multiple scales and tune using the resulting information
- Deploy the system with performance logging features.

ftp://ftp.ora.com/pub/examples/java/javapt/technique-list.html
A long list of most of the tuning techniques covered in my "Java Performance Tuning" book (Page last updated August 2000, Added 2000-10-23, Author Jack Shirazi, Publisher O'Reilly). Tips:

- [Since the referred to page is already a summary list, I have not extracted it here. Especially since there are nearly 300 techniques listed. Check the page out directly].

http://www.onjava.com/pub/a/onjava/2001/05/30/optimization.html
Comparing the performance of LinkedLists and ArrayLists (and Vectors) (Page last updated May 2001, Added 2001-06-18, Author Jack Shirazi, Publisher OnJava). Tips:

- ArrayList is faster than Vector except when there is no lock acquisition required in HotSpot JVMs (when they have about the same performance).
- Vector and ArrayList implementations have excellent performance for indexed access and update of elements, since there is no overhead beyond range checking.
- Adding elements to, or deleting elements from the end of a Vector or ArrayList also gives excellent performance except when the capacity is exhausted and the internal array has to be expanded.
- Inserting and deleting elements to Vectors and ArrayLists always require an array copy (two copies when the internal array must be grown first). The number of elements to be copied is proportional to [size-index], i.e. to the distance between the insertion/deletion index and the last index in the collection. The array copying overhead grows significantly as the size of the collection increases, because the number of elements that need to be copied with each insertion increases.
- For insertions to Vectors and ArrayLists, inserting to the front of the collection (index 0) gives the worst performance, inserting at the end of the collection (after the last element) gives the best performance.
- LinkedLists have a performance overhead for indexed access and update of elements, since access to any index requires you to traverse multiple nodes.
- LinkedList insertions/deletion overhead is dependent on the how far away the insertion/deletion index is from the closer end of the collection.
- Synchronized wrappers (obtained from Collections.synchronizedList(List)) add a level of indirection which can have a high performance cost.
- Only List and Map have efficient thread-safe implementations: the Vector and Hashtable classes respectively.
- List insertion speed is critically dependent on the size of the collection and the position where the element is to be inserted.
- For small collections ArrayList and LinkedList are close in performance, though ArrayList is generally the faster of the two. Precise speed comparisons depend on the JVM and the index where the object is being added.
- Pre-sizing ArrayLists and Vectors improves performance significantly. LinkedLists cannot be pre-sized.
- ArrayLists can generate far fewer objects for the garbage collector to reclaim, compared to LinkedLists.
- For medium to large sized Lists, the location where elements are to inserted is critical to the performance of the list. ArrayLists have the edge for random access.
- A dedicated List implementation designed to match data, collection types and data manipulation algorithms will always provide the best performance.
- ArrayList internal node traversal from the start to the end of the collection is significantly faster than LinkedList traversal. Consequently queries implemented in the class can be faster.
- Iterator traversal of all elements is faster for ArrayList compared to Linkedlist.

http://www.onjava.com/pub/a/onjava/2001/07/09/optimization.html
Using the WeakHashMap class (Page last updated June 2001, Added 2001-07-20, Author Jack Shirazi, Publisher OnJava). Tips:

- WeakHashMap can be used to reduce memory leaks. Keys that are no longer strongly referenced from the application will automatically make the corresponding value reclaimable.
- To use WeakHashMap as a cache, the keys that evaluate as equal must be recreatable.
- Using WeakHashMap as a cache gives you less control over when cache elements are removed compared with other cache types.
- Clearing elements of a WeakHashMap is a two stage process: first the key is reclaimed, then the corresponding value is released from the WeakHashMap.
- String literals and other objects like Class which are held directly by the JVM are not useful as keys to a WeakHashMap, as they are not necessarily reclaimable when the application no longer references them.
- The WeakHashMap values are not released until the WeakHashMap is altered in some way. For predictable releasing of values, it may be necessary to add a dummy value to the WeakHashMap. If you do not call any mutator methods after populating the WeakHashMap, the values and internal WeakReference objects will never be dereferenced [no longer true from 1.4, where most methods now allow values to be released].
- WeakHashMap wraps an internal HashMap adding an extra level of indirection which can be a significant performance overhead. [no longer true from 1.4].
- Every call to get() creates a new WeakReference object. [no longer true from 1.4].
- WeakHashMap.size() iterates through the keys, making it an operation that takes time proportional to the size of the WeakHashMap. [no longer true from 1.4].
- WeakHashMap.isEmpty() iterates through the collection looking for a non-null key, so a WeakHashMap which is empty requires more time for isEmpty() to return than a similar WeakHashMap which is not empty. [no longer true from 1.4, where isEmpty() is now slower than previous versions].

http://java.oreilly.com/news/jptsummary_1100.html
ftp://ftp.ora.com/pub/examples/java/javapt/summary.html
A high level overview of technical performance tuning, covering 5 levels of tuning competence. (Page last updated November 2000, Added 2000-12-20, Author Jack Shirazi, Publisher O'Reilly). Tips:

- Start tuning by examining the application architecture for potential bottlenecks.
- Architecture bottlenecks are often easy to spot: they are the connecting lines on the diagrams; the single threaded components; the components with many connecting lines attached; etc.
- Ensure that application performance is measureable for the given performance targets.
- Ensure that there is a test environment which represents the running system. This test-bed should support testing the application at different loads, including a low load and a fully scaled load representing maximum expected usage.
- After targeting design and architecture, the biggest bang for your buck in terms of improving performance is choosing a better VM, and then choosing a better compiler.
- Start code tuning with proof of concept bottleneck removal: this consists of using profilers to identify bottlenecks, then making simplified changes which may only improve the performance at the bottleneck for a specialized set of activities, and proceeding to the next bottleneck. After tuning competence is gained, move to full tuning.
- Each multi-user performance test can typically take a full day to run and analyse. Even simple multi-user performance tuning can take several weeks.

- After the easily idenitified bottlenecks have been removed, the remaining performance improvements often come mainly from targeting loops, structures and algorithms.
- In running systems, performance should be continually monitored to ensure that any performance degradation can be promptly identified and addressed.

http://www.oreilly.com/catalog/javapt/chapter/ch04.html
Chapter 4 of "Java Performance Tuning", "Object Creation". (Page last updated September 2000, Added 2000-10-23, Author Jack Shirazi, Publisher O'Reilly). Tips:

- Establish whether you have a memory problem.
- Reduce the number of temporary objects being used, especially in loops.
- Avoid creating temporary objects within frequently called methods.
- Presize collection objects.
- Reuse objects where possible.
- Empty collection objects before reusing them. (Do not shrink them unless they are very large.)
- Use custom conversion methods for converting between data types (especially strings and streams) to reduce the number of temporary objects.
- Define methods that accept reusable objects to be filled in with data, rather than methods that return objects holding that data. (Or you can return immutable objects.)
- Canonicalize objects wherever possible. Compare canonicalized objects by identity. [Canonicalizing objects means having only a single reference of an object, with no copies possible].
- Create only the number of objects a class logically needs (if that is a small number of objects).
- Replace strings and other objects with integer constants. Compare these integers by identity.
- Use primitive data types instead of objects as instance variables.
- Avoid creating an object that is only for accessing a method.
- Flatten objects to reduce the number of nested objects.
- Preallocate storage for large collections of objects by mapping the instance variables into multiple arrays.
- Use `StringBuffer` rather than the string concatenation operator (+).
- Use methods that alter objects directly without making copies.
- Create or use specific classes that handle primitive data types rather than wrapping the primitive data types.
- Consider using a `ThreadLocal` to provide threaded access to singletons with state.
- Use the `final` modifier on instance-variable definitions to create immutable internally accessible objects.
- Use `WeakReferences` to hold elements in large canonical lookup tables. (Use `SoftReferences` for cache elements.)
- Reduce object-creation bottlenecks by targeting the object-creation process.
- Keep constructors simple and inheritance hierarchies shallow.
- Avoid initializing instance variables more than once.
- Use the `clone()` method to avoid calling any constructors.
- Clone arrays if that makes their creation faster.
- Create copies of simple arrays faster by initializing them; create copies of complex arrays faster by cloning them.
- Eliminate object-creation bottlenecks by moving object creation to an alternative time.
- Create objects early, when there is spare time in the application, and hold those objects until required.
- Use lazy initialization when there are objects or variables that may never be used, or when you need to distribute the load of creating objects.

- Use lazy initialization only when there is a defined merit in the design, or when identifying a bottleneck which is alleviated using lazy initialization.

http://java.oreilly.com/news/javaperf_0900.html
My article on basic optimizations for queries on collections (Page last updated September 2000, Added 2000-10-23, Author Jack Shirazi, Publisher O'Reilly). Tips:

- Use short-circuit boolean operators instead of the normal boolean operators.
- Eliminate any unnecessarily repeated method calls from loops.
- Eliminate unnecessary casts.
- Avoid synchronization where possible.
- Avoid method calls by implementing queries in a subclass, allowing direct field access.
- Use temporary local variables to manipulate data fields (instance/class variables).
- Use more precise object typing where possible.
- Before manual tuning, HotSpot VMs are often faster than JIT VMs. But JIT VMs tend to benefit more from manual tuning and can end up faster than HotSpot VMs.

http://www.javaworld.com/javaworld/jw-11-2000/jw-1117-optimize.html
Article about optimizing queries on Maps. (Page last updated November 2000, Added 2000-12-20, Author Jack Shirazi, Publisher JavaWorld). Tips:

- Avoid using synchronization in read-only or single-threaded queries.
- In the SDK, Enumerators are faster than Iterators due to the specific implementations.
- Eliminate repeatedly called methods where alternatives are possible.
- Iterator.hasNext() and Enumerator.hasMoreElements() do not need to be repeatedly called when the size of the collection is known. Use collection.size() and a loop counter instead.
- Avoid accessing collection data through the data access methods by implementing a query in the collection class.
- Elminate repeated casts by casting once and holding the cast item in a correctly typed variable.
- Reimplement the collection class to specialize for the data being held in the collection.
- Reimplment the Map class to use a hash function which is more efficient for the data being mapped.

http://www.onjava.com/pub/a/onjava/2001/01/25/hash_functions.html
Optimizing hash functions: generating a perfect hash function (Page last updated January 2001, Added 2001-02-21, Author Jack Shirazi, Publisher OnJava). Tips:

- *perfect* hash functions guarantee that every key maps to a separate entry in a hashtable, and so provide more efficient hastable implementations than generic hash functions.
- *perfect* hash functions are possible when the key data is restricted to a known set of elements.
- Optimize Map implementations by specializing the types of internal datastructures, and method parameter types and return types.
- Optimize Map implementations by using a specialized hash function that is optimized for the key type, rather than generic to all possible types of keys.
- Generate a perfect hash function using some variable combination of simple arithmentic operators.
- Perfect hash functions may require excessive amounts of memory.
- Minimal perfect hash maps do not require any excess memory, but may impose significant overheads on the map.

http://www.onjava.com/pub/a/onjava/2002/03/20/optimization.html
Microtuning (Page last updated March 2002, Added 2002-03-25, Author Jack Shirazi, Publisher OnJava). Tips:

- Performance is dependent on data as well as code. Different data can make identical code perform very differently.
- Always start tuning with a baseline measurement.
- The System.currentTimeMillis() method is the most basic measuring tool for tuning.
- You may need to repeatedly call a method in order to reliably measure its average execution time.
- Minimize the possibility that CPU time will be allocated to anything other than the test while it is running by ensuring no other processes are runing during the test, and that the test remains in the foreground.
- Baseline measurements normally show some useful information, e.g. the average execution time for one call to a method.
- Multiplying the average time taken to execute a method or sequence of methods, by the number of times that sequence will be called in a time period, gives you an estimate of the fraction of the total time that the sequence takes.
- There are three routes to tuning a method: Consider unexpected differences in different test runs; Analyze the algorithm; Profile the method.
- Creating an exception is a costly procedure, because of filling in stack trace.
- A profiler should ideally be able to take a snapshot of performance between two arbitrary points.
- Tuning is an iterative process: you normally find one bottleneck, make changes that improve performance, test those changes, and then start again.
- Algorithm changes usually provide the best speedup, but can be difficult to find.
- Examining the code for the causes of the differences in speed between two variations of test runs can be useful, but is restricted to those tests for which you can devise alternatives that show significant timing variations.
- Profiling is always an option and almost always provides something that can be speeded up. But the law of diminishing returns kicks in after a while, leaving you with bottlenecks that are not worth speeding up, because the potential speedup is too small for the effort required.
- Generic integer parsing (as with the Integer constructors and methods) may be overkill for converting simple integer formats.
- Simple static methods are probably best left to be inlined by the JIT compiler rather than by hand.
- String.equals() is expensive if you are only testing for an empty string. It is quicker to test if the length of the string is 0.
- Set a target speedup to reach. With no target, tuning can carry on for much longer than is needed.
- A generic tuning procedure is: Identify the bottleneck; Set a performance target; Use representative data; Measure the baseline; Analyze the method; Test the change; Repeat.

http://www.onjava.com/pub/a/onjava/2000/12/15/formatting_doubles.html
Efficiently formatting doubles (Page last updated December 2000, Added 2000-12-20, Author Jack Shirazi, Publisher OnJava). Tips:

- Double.toString(double) is slow. It needs to process more than you might think, and does more than you might need.
- Proprietary conversion algorithms can be significantly faster. One such algorithm is presented in the article.

- Converting integers to strings can also be faster than the SDK. An algorithm successively stripping off the highest is used in the article.
- Formatting numbers using java.text.DecimalFormat is always slower than Double.toString(double), because it first calls Double.toString(double) then parses and converts the result.
- Formatting using a proprietary conversion algorithm can be faster than any of the methods discussed so far, if the number of digits being printed is not large. The actual time taken depends on the number of digits being printed.

http://www.onjava.com/pub/a/onjava/2001/09/25/optimization.html
Multiprocess JVMs (Page last updated September 2001, Added 2001-10-22, Author Jack Shirazi, Publisher OnJava). Tips:

- Using or implementing a multiprocess framework to combine Java processes into one JVM can save on memory space overheads and reduce startup time.

http://www.onjava.com/pub/a/onjava/2001/12/05/optimization.html
Measuring JDBC performance (Page last updated December 2001, Added 2001-12-26, Author Jack Shirazi, Publisher OnJava). Tips:

- Effectively profiling distributed applications can be difficult. I/O can show up as significant in profiling, simply because of the nature of a distributed application.
- It can be unclear whether threads blocking on reads and writes are part of a significant bottleneck or simply a side issue.
- When profiling, it is usually worthwhile to have separate measurements available for the communication subsystems.
- Wrapping the JDBC classes provides an effective technique for measuring database calls.
- [Article discusses how to create JDBC wrapers to measure the performance of database calls].
- If more than a few rows of a query are being read, then the ResultSet.next() method can spend a significant amount of time fetching rows from the database, and this time should be included in measurements of database access.
- JDBC wrappers are simple and robust, and require very little alteration to the application using them (i.e, are low maintenance), so they are suitable to be retained within a deployed application.

http://www.onjava.com/pub/a/onjava/2001/08/22/optimization.html
Catching OutOfMemoryErrors (Page last updated August 2001, Added 2001-10-22, Author Jack Shirazi, Publisher OnJava). Tips:

- -Xmx and -Xms (-mx and -ms) specify the heap max and starting sizes. Runtime.totalMemory() gives the current process size, Runtime.maxMemory() (available from SDK 1.4) gives the -Xmx value.
- Repeatedly allocating memory by creating objects and holding onto them will expand the process to its maximum possible size. This technique can also be used to flush memory.
- If a process gets too large, the operating system will start paging the process causing a severe decrease in performance.
- It is reasonable to catch the OutOfMemoryError if you can restore your application to a known state that can proceed with processing. For example, daemon service threads can often do this.

http://www.onjava.com/pub/a/onjava/2001/10/23/optimization.html
The RandomAccess interface. (Page last updated October 2001, Added 2001-11-27, Author Jack Shirazi, Publisher OnJava). Tips:

- A java.util.List object which implements RandomAccess should be faster when using List.get() than when using Iterator.next().
- Use `instanceof RandomAccess` to test whether to use List.get() or Iterator.next() to traverse a List object.
- [Article describes how to guard the test to support all versions of Java].

http://www.cs.berkeley.edu/~mdw/proj/java-nbio/
Whoopee!! A non-blocking I/O library for Java. This is the single most important functionality missing from the SDK for scalable server applications. The important class is SelectSet which allows you to multiplex all your i/o streams. If you want a scalable server and can use this class then DO SO. *NOTE THAT SDK 1.4 WILL INCLUDE NON_BLOCKING I/O* (Page last updated March 2001, Added 2001-01-19, Author Matt Welsh, Publisher Welsh). Tips:

- [The system select(2)/poll(2) functions allow you to take any collection of i/o streams and ask the operating system to check whether any of them can execute read/write/accept without blocking. The system call will block if requested until any one of the i/o streams is ready to execute. Before Java, no self-respecting server would sit on multiple threads in blocked i/o mode, wasting thread resources: instead select/poll would have been used.]

http://www.cs.cmu.edu/~jch/java/optimization.html
For years, Jonathan Hardwick's old but classic site was the only coherent Java performance tuning site on the web. He built it while doing his PhD. It wasn't updated beyond March 1998, when he moved to Microsoft, but most tips are still useful and valid. The URL is for the top page, there are another eight pages. Thanks Jonathan. (Page last updated March 1998, Added 2000-10-23, Author Jonathan Hardwick, Publisher Hardwick). Tips:

- Don't optimize as you go. Write your program concentrating on clean, correct, and understandable code.
- Use profiling to find out where that 80% of execution time is going, so you know where to concentrate your effort.
- Always run "before" and "after" benchmarks.
- Use the right algorithms and data structures.
- Compile with optimization flag, javac -O.
- Use a JIT.
- Multithread for multi-processor machines.
- Use clipping to reduce the amount of work done in repaint()
- Use double buffering to improve perceived speed.
- Use image strips or compression to speed up downloading times.
- Animation in Java Applets from JavaWorld and Performing Animation from Sun are two good tutorials.
- Use high-level primitives; it's much faster to call drawPolygon() on a bunch of points than looping with drawLine().
- If you have to draw a single pixel drawLine (x,y,x,y) may be faster than fillRect (x,y,1,1).
- Use Buffered I/O classes.
- Avoid synchronized methods if you can.
- Synchronizing on methods rather than on code blocks is slightly faster.
- Use exceptions only where you really need them.
- Use StringBuffer instead of +.

- Use System.arraycopy() and any other optimized API's available from the SDK.
- Replace the generic standard classes with faster implementations specific to the application.
- Create subclasses to override methods with faster versions.
- Avoid expensive constructs and data structures, e.g. one-dimensional array is faster than a two-dimensional array.
- Use the faster switch bytecode.
- Use private and static methods, and final classes, to encourage inlining by the compiler.
- Reuse objects.
- Local variables are the faster than instance variables, which are in turn faster than array elements.
- ints are the fastest data type.
- Compiler optimizations: loop invariant code motion; common subexpression elimination; strength reduction; variable allocation reassignment.
- Use java -prof or other profiler.
- Use a timing harness to run benchmarks.
- Use a memory measurement harness to run benchmarks.
- Call system.gc() before every timing run to minimize inconsistent results due to garbage collection in the middle of a run.
- Use JAR or zip files.
- If size is a constraint: use SDK classes wherever possible; inherit whatever possible; put common code in one place; initialize big arrays at runtime by parsing a string; use short names;

http://www.ddjembedded.com/resources/articles/2001/0112g/0112g.htm
Balancing Network Load with Priority Queues (Page last updated December 2001, Added 2002-02-22, Author Frank Fabian, Publisher Dr. Dobb's). Tips:

- Hardware traffic managers redirect user requests to a farm of servers based on server availability, IP address, or port number. All traffic is routed to the load balancer, then requests are fanned out to servers based on the balancing algorithm.
- Popular load-balancing algorithms include: server availability (find a server with available processing capability); IP address management (route to the nearest server by IP address); port number (locate different types of servers on different machines, and route by port number); HTTP header checking (route by URI or cookie, etc).
- Web hits should cater for handling peak hit rate, not the average rate.
- You can model hit rates using gaussian distribution to determine the average hit rate per time unit (e.g. per second) at peak usage, then a poisson probability gives the probability of a given number of users simulatneously hitting the server within that time unit. [Article gives an example with gaussian fitted to peak traffic of 4000 users with a standard deviation of 20 minutes resulting in an average of 1.33 users per second at the peak, which in turn gives the probabilities that 0, 1, 2, 3, 4, 5, 6 users hitting the server within one second as 26%, 35%, 23%, 10%, 3%, 1%, 0.2%. Service time was 53 milliseconds, which means that the server can service 19 hits per second without the service rate requiring requests being queued.]
- System throughput is the arrival rate divided by the service rate. If the ratio becomes greater than one, requests exceed the system capability and will be lost or need to be queued.
- If requests are queued because capacity is exceeded, the throughput must drop sufficiently to handle the queued requests or the system will fail (the service rate must increase or arrival rate decrease). If the average throughput exceeds 1, then the system will fail.
- Sort incoming requests into different priority queues, and service the requests according to the priorities assigned to each queue. [Article gives the example where combining user and

automatic requests in one queue can result in a worst case user wait of 3.5 minutes, as opposed to less than 0.1 seconds if priority queues are used].
- [Note that Java application servers often do not show a constant service time. Instead the service time often increases with higher concurrency due to non-linear effects of garbage collection].

http://library.cs.tuiasi.ro/programming/java/cutting_edge_java_game_programming/ewtoc.html
"Cutting Edge Java Game Programming". Oldish but still useful intro book to games programming using Java. (Page last updated 1996, Added 2001-06-18, Author Neil Bartlett, Steve Simkin , Publisher Coriolis). Tips:

- AWT components are not useful as game actors (sprites) as they do not overlap well, nor are they good at being moved around the screen.
- Celled image files efficiently store an animated image by dividing an image into a rectangular grid of cells, and allocating a different animation image to each cell. A sequence of similar images (as you would have for an animation) will be stored and transferred efficiently in most image formats.
- Examining pixels using PixelGrabber is slow.
- drawImage() can throw away and re-load images in response to memory requirements, which can make things slow.
- Pre-load and pre-scale images before using them to get a smoother and faster display.
- The more actors (sprites), the more time it takes to draw and the slower the game appears.
- Use double-buffering to move actors (sprites), by redrawing the actor and background for the relevant area.
- Redraw speed depends on: how quickly each object is drawn; how many objects are drawn; how much of each object is drawn; the total number of drawing operations. You need to reduce some or all of these until you get to about 30 redraws per second.
- Don't draw actors or images that cannot be seen.
- If an actor is not moving then incorporate the actor as part of the background.
- Only redraw the area that has changed, e.g. the old area where an actor was, and the new area where it is. Redrawing several small areas is frequently faster than drawing one large area. For the redraws, eliminate overlapping areas and merge adjacent (close) areas so that the number of redraws is kept to a minimum.
- Put slow and fast drawing requirements in separate threads.
- Bounding-box detection can use circles for the bounding box which requires a simple radii detection.
- Load sounds in a background thread.
- Make sure you have a throttle control that can make the game run slower (or pause) when necessary.
- The optimal network topology for network games depends on the number of users.
- If the cumulative downloading of your applet exceeds the player?s patience, you?ve lost a customer.
- The user interface should always be responsive. A non-responsive window means you will lose your players. Give feedback on necessary delays. Provide distractions when unavoidable delays will be lengthy [more than a few seconds].
- Transmission time varies, and is always slow compared to operations on the local hardware. You may need to decide the outcome of the action locally, then broadcast the result of the action. This may require some synchronization resolution.
- Latency between networked players can easily lead to de-synchronized action and player frustration. Displays should locally simulate remote action as continuing current activities/motions, until the display is updated. On update, the actual current situation should be smoothly resolved with the simulated current situation.

- Sending activity updates more frequently ensures smoother play and better synchronization between networked players, but requires more CPU effort and so affects the local display. In order to avoid adversely affecting local displays, send actvity updates from a low priority thread.
- Discard any out-of-date updates: always use the latest dated update.
- A minimum broadcast delay of one-third the average network connection travel time is appropriate. Once you exceed this limit, the additional traffic can cause more grief than benefit.
- Put class files into a (compressed) container for network downloading.
- Avoid repeatedly evaluating invariant expressions in a loop.
- Take advantage of inlining where possible (using final, private and static keywords, and compiling with javac -O)
- Profile the code to determine the expensive methods (e.g. using the -prof option)
- Use a dissassembler (e.g. like javap) to determine which of various alternative coding formulations produces smaller bytecode.
- To reduce the number of class files and their sizes: use the SDK classes as much as possible; and implement common functionality in one place only.
- To optimize speed: avoid synchronized methods; use buffered I/O; reuse objects; avoid unnecessary screen painting.
- Raycasting is faster than raytracing. Raycasting maps 2D data into a 3D world, drawing entire vertical lines using one ray. Use precalculated values for trignometric and other functions, based on the angle increments chosen for your raycasting.
- In the absence of a JIT, the polygon drawing routines fron the AWT are relatively efficient (compared to array manipulation) and may be faster than texture mapping.
- Without texture mapping, walls can be drawn faster with one call to fillPolygon (rather than line by line).
- An exponential jump search algorithm can be used to reduce ray casts - by quickly finding boundaries where walls end (like a binary search, but double increments until your overshoot, then halving increments from the last valid wall position).
- It is usually possible to increase performance at the expense of image quality and accuracy. Techniques include reducing pixel depth or display resolution, field interlacing, aliasing. The key, however, is to degrade the image in a way that is likely to be undetectable or unnoticeable to the user. For example a moving player often pays less attention to image quality than a resting or static player.
- Use information gathered during the rendering of one frame to approximate the geometry of the next frame, speeding up its rendering.
- If the geometry and content is not too complicated, binary space partition trees map the view according to what the player can see, and can be faster than ray casting.

http://www.javaworld.com/javaworld/jw-03-2001/jw-0323-performance.html
Designing remote interfaces (Page last updated March 2001, Added 2001-04-20, Author Brian Goetz, Publisher JavaWorld). Tips:

- Remote object creation has overheads: several objects needed to support the remote object are also created and manipulated.
- Remote method invocations involve a network round-trip and marshalling and unmarshaling of parameters. This adds together to impose a significant latency on remote method invocations.
- Different object parameters can have very different marshalling and unmarshaling costs.
- A poorly designed remote interface can kill a program's performance.
- Excessive remote invocation network round-trips are a huge performance problem.

- Calling a remote method that returns multiple values contained in a temporary object (such as a Point), rather than making multiple consecutive method calls to retrieve them individually, is likely to be more efficient. (Note that this is exactly the opposite of the advice offered for good performance of local objects.)
- Avoid unnecessary round-trips: retrieve several related items simultaneously in one remote invocation, if possible.
- Avoid returning remote objects when the caller may not need to hold a reference to the remote object.
- Avoid passing complex objects to remote methods when the remote object doesn't necessarily need to have a copy of the object.
- If a common high-level operation requires many consecutive remote method calls, you need to revisit the class's interface.
- A naively designed remote interface can lead to an application that has serious scalability and performance problems.
- [Article gives examples showing the effect of applying the listed advice].

http://www.glenmccl.com/jperf/
Glen McCluskey's paper with 30 tuning tips, now free. (Page last updated October 1999, Added 2000-10-23, Author Glen McCluskey, Publisher McCluskey). Tips:

- Faster algorithms are better.
- Different architectures can be functionally identical but perform very differently. Keep performance in mind at the design stage.
- Use the fastest available JVM.
- Use static variables for fields that only need to be assigned once.
- Reuse objects where reasonable, e.g. nodes of a linked list.
- Inline methods manually where appropriate. [Better to use a preprocessor].
- Keep methods short and simple to make them automatic inlining candidates.
- `final` classes can be faster.
- Synchronized methods are slower than the identical non-synchronized one.
- Consider using non-synchronized classes and synchronized-wrappers.
- Access to private members of inner classes from the enclosing class goes by a method call even if not intended to.
- Use StringBuffer instead of the '+' String concatentation operator.
- Use `char[]` arrays directly to create Strings rather than StringBuffers.
- '==' is faster than equals().
- intern() Strings to enable identity (==) comparisons.
- Convert strings to char[] arrays to process characters, rather than accessing characters one at a time using String.charAt().
- Creating Doubles from strings is slow.
- Buffer i/o.
- MessageFormat is slow.
- Reuse objects.
- File information such as File.length() requires a system call and can be slow.
- Use System.arraycopy() to copy arrays.
- ArrayList is faster than Vector.
- Preset array capacity to as large as will be required.
- LinkedList is faster than ArrayList for inserting elements to the front of the array, but slower at indexed lookup.
- Program using interfaces so that the actual structure can be easily swapped to improve performance.
- Use the -g:none option to the javac compiler.

- Primitive data wrapper classes (e.g. Integer) are slower than using the primitive data directly.
- Null out references when they are no longer used so that garbage collection can reclaim their space.
- Use SoftReferences to recycle memory when required.
- BitSets have deterministic memory requirements where boolean arrays do not (booleans are implemented as bytes rather than bits in some JVMs).
- Use sparse arrays to hold widely spaced indexable data.

http://www.sun.com/solaris/java/wp-java/6.html
Performance tuning part of a white paper about Java on Solaris 2.6. (Page last updated 2000, Added 2000-10-23, Author ?, Publisher Sun). Tips:

- To profile I/O calls, use a profiler or use truss and look for read() and write() system calls.
- Buffer I/O. Tune the buffer size (bigger is usually better if memory is available).
- Use char arrays for all character processing in loops, rather than using the String or StringBuffer classes.
- Avoid character processing using methods (e.g. charAt(), setCharAt()) inside a loop.
- Set the initial StringBuffer size to the maximum string length, if it is known.
- StringTokenizer is very inefficient, and can be optimized by storing the string and delimiter in a character array instead of in String, or by storing the highest delimiter character to allow a quicker check.
- Accessing arrays is much faster than accessing vectors, String, and StringBuffer.
- Use System.arraycopy() to improve performance.
- Vector is convenient to use, but inefficient. Ensure that elementAt() is not used inside a loop.
- FastVector is faster than Vector by making the elementData field public, thus avoiding (synchronized) calls to elementAt().
- Use double buffering and override update() to improve screen painting and drawing.
- Use custom LayoutManagers.
- Repaint only the damaged regions (use ClipRect).
- To improve image handling: use MediaTracker; use your own imageUpdate() method; pre-decode and store the image in an array - image decoding time is greater than loading time. Pre-decoding using PixelGrabber and MemoryImageSource should combine multiple images into one file for maximum speed.
- Increase the initial heap size from the 1-MByte default with -ms and -mx [-Xms and -Xmx].
- Use -verbosegc.
- Take size into account when allocating arrays (for instance, if short is big enough, use it instead of int.
- Avoid allocating objects in loops (readLine() is a common example).
- Minimize synchronization.
- Polling is only acceptable when waiting for outside events and should be performed in a "side" thread. Use wait/notify instead.
- Move loop invariants outside the loop.
- Make tests as simple as possible.
- Perform the loop backwards (this actually performs slightly faster than forward loops do). [Actually it is converting the test to compare against 0 that makes the difference].
- Use only local variables inside a loop; assign class fields to local variables before the loop.
- Move constant conditionals outside loops.
- Combine similar loops.
- Nest the busiest loop, if loops are interchangeable.
- Unroll the loop, as a last resort.

- Convert expressions to table Lookups.
- Use caching.
- Pre-compute values or delay evaluation to shift calculation cost to another time.
- [Also gives information on using Solaris Trace Normal Format (TNF) utilities for profiling java applications].

http://www.javareport.com/html/from_pages/article.asp?id=252
Detailed article on load testing systems (Page last updated January 2001, Added 2001-01-19, Author Himanshu Bhatt, Publisher Java Report). Tips:

- Internet systems should be load-tested throughout development.
- Load testing can provide the basis for: Comparing varying architectural approaches; Performance tuning; Capacity planning.
- Initially you should identify the probable performance and scalability based on the requirements. You should be asking about: numbers of users/components; component interactions; throughput and transaction rates; performance requirements.
- Factor in batch requirements and performance characteristics of dependent (sub)systems. Note that additional layers, like security, add overheads to performance.
- Logging and stateful EJB can degrade performance.
- After the initial identification phase, the target should be for a model architecture that can be load-tested to feedback information.
- Scalability hotspots are more likely to exist in the tiers that are shared across multiple client sessions.
- Performance measurements should be from presentation start to presentation completion, i.e. user clicks button (start) and information is displayed (completion).
- Use load-test suites and frameworks to perform repeatable load testing.

http://www.devx.com/free/articles/2000/maso01/maso01-1.asp
Article on using syslog to track performance across distributed systems (Page last updated December 2000, Added 2001-01-19, Author Brian Maso, Publisher DevX). Tips:

- Use syslog to log distributed system performance.
- Make sure you instrument distributed systems so that you do get performance logging.

http://www.as400.ibm.com/developer/java/topics/jdbctips.html
JDBC Performance Tips (targeted at AS/400, but generically applicable) (Page last updated February 2001, Added 2001-03-21, Authors Richard Dettinger and Mark Megerian, Publisher IBM). Tips:

- Move to the latest releases of Java as they become available.
- Use prepared statements (PreparedStatement class) [article provides coded example of using Statement vs. PreparedStatement].
- Note that two database calls are made for each row in a ResultSet: one to describe the column, the second to tell the db where to put the data. PreparedStatements make the description calls at construction time, Statements make them on every execution.
- Avoid retrieving unnecessary columns: don't use "SELECT *".
- If you are not using stored procedures or triggers, turn off autocommit. All transaction levels operate faster with autocommit turned off, and doing this means you must code commits. Coding commits while leaving autocommit on will result in extra commits being done for every db operation.
- Use the appropriate transaction level. Increasing performance costs for transaction levels are: TRANSACTION_NONE; TRANSACTION_READ_UNCOMMITTED;

TRANSACTION_READ_COMMITTED; TRANSACTION_REPEATABLE_READ; TRANSACTION_SERIALIZABLE. Note that TRANSACTION_NONE, with autocommit set to true gives access to triggers, stored procedures, and large object columns.
- Store string and char data as Unicode (two-byte characters) in the database.
- Avoid expensive database query functions such as: getBestRowIdentifier; getColumns; getCrossReference; getExportedKeys; getImportedKeys; getPrimaryKeys; getTables; getVersionColumns.
- Use connection pooling, either explicitly with your own implementation, or implicitly via a product that supports connection pooling.
- Use blocked fetchs (fetching table data in blocks), and tailor the block size to reduce calls to the database, according to the amount of data required.
- Use batch updates (sending multiple rows to the database in one call).
- Use stored procedures where appropriate. These benefit by reducing JDBC complexity, are faster as they use static SQL, and move execution to the server and potentially reduce network trips.
- Use the type-correct get() method, rather than getObject().

http://www.patrick.net/jpt/index.html
Patrick Killelea's Java performance tips. (Page last updated 1999, Added 2000-10-23, Author Patrick Killelea, Publisher Killelea). Tips:

- System.currentTimeMillis may take up to 0.5 milliseconds to execute.
- The architecture and algorithms of your program are much more important than any low-level optimizations you might perform.
- Tune at the highest level first.
- Make the common case fast (Amdahl's advice).
- Use what you know about the runtime platform or usage patterns.
- Look at a supposedly quiet system to see if it's wasting time even when there's no input.
- Keep small inheritance chains.
- Use stack (local) variables in preference to class variables.
- Merge classes.
- drawPolygon() is faster than using drawLine() repeatedly.
- Don't create too may objects.
- Reuse objects if possible.
- Beware of object leaks (references to objects that are never nulled).
- Accessor methods increase overhead.
- Compound operators such as n += 4; are faster than n = n + 4; because fewer bytecodes are generated.
- Shifting by powers of two is faster than multiplying.
- Multiplication is faster than exponentiation.
- int increments are faster than byte or short increments.
- Floating point increments are much slower than any integral increment.
- Memory access from better to worse: local vars; supersuperclass instance variable; superclass instance var; class instance var; class static var; array elements.
- It can help to copy slower-access vars to fast local vars if you are going to operate on them repeatedly, as in a loop.
- Use networking timeouts, TCP_NODELAY, SO_TIMEOUT, especially in case of dying DNS servers.
- Buffer network io. [or read explicitly in chunks].
- Avoid reverse DNS where you can.
- Use UDP rather than TCP if speed is more important than accuracy.

- Use threads. Prioritize threads. Use notify instead of notifyAll. Use synchronization sparingly.
- Counting down is often faster than counting up. [the loop test comparison to 0 is what matters].
- Keep synchronized methods out of loops if you possibly can.
- Avoid excessive String manipulation.
- Use String Buffers or Arrays rather than String.
- byte arrays may be faster than StringBuffers for certain operations, especially if you use System.arraycopy().
- Use StringBuffer rather than the + operator.
- Watch out for slow fonts, Fonts vary in speed of rendering.
- Keep the paint method small. It will get called a lot.
- Double buffer where possible.
- For some applications that access the date a lot, it can help to set the local timezone to be GMT, so that no conversion has to take place.
- Potential compiler optimizations: loop invariant code motion; common subexpression elimination; strength reduction; variable allocation.
- Don't turn off native threads.
- Use .jar files.
- Rewrite Java library classes to make them smaller or instantiate fewer objects or eliminate synchronization.
- Install classes locally.

http://java.sun.com/docs/books/tutorial/extra/fullscreen/
Tutorial on the full screen capabilities in the 1.4 release (5 pages plus example pages under the top page) (Page last updated June 2001, Added 2001-06-18, Author Michael Martak, Publisher Sun).
Tips:

- The full-screen exclusive mode provides maximum image display and drawing performance by allowing direct drawing to the screen.
- Use java.awt.GraphicsDevice.isFullScreenSupported() to determine if full-screen exclusive mode is available. If it is not available, full-screen drawing can still be used, but better performance will be obtained by using a fixed size window in normal screen mode. Full-screen exclusive applications should not be resizable.
- Turn off decoration using the setUndecorated() method.
- Change the screen display mode (size, depth and refresh rate), to the best match for your image bit depth and display size so that scaling and other image alterations can be avoided or minimized.
- Don't define the screen painting code in the paint() method called by the AWT thread. Define your own rendering loop for screen drawing, to be executed in any thread other than the AWT thread.
- Use the setIgnoreRepaint() method on your application window and components to turn off all paint events dispatched from the operating system completely, since these may be called during inappropriate times, or worse, end up calling paint, which can lead to race conditions between the AWT event thread and your rendering loop.
- Do not rely on the update or repaint methods for delivering paint events.
- Do not use heavyweight components, since these will still incur the overhead of involving the AWT and the platform's windowing system.
- Use double buffering (drawing to an off-screen buffer, then copying the finished drawing to the screen).
- Use page-flipping (changing the video pointer so that an off-screen buffer becomes the on-screen buffer, with no image copying required).

- Use a flip chain (a sequence of off-screen buffers which the video pointer successively points to one after the other).
- java.awt.image.BufferStrategy provides getDrawGraphics() (to get an off-screen buffer) and show() (to display the buffer on screen).
- Use java.awt.BufferCapabilities to customize the BufferStrategy for optimizing the performance of your application.
- If you use a buffer strategy for double-buffering in a Swing application, you probably want to turn off double-buffering for your Swing components,
- Multi-buffering is only useful when the drawing time exceeds the time spent to do a show.
- Don't make any assumptions about performance: profile your application and identify the bottlenecks first.

http://www.devresource.hp.com/JavaATC/JavaPerfTune/index.html
HP Java tuning site, including optimizing Java and optimizing HPUX for Java. This is the top page, but several useful pages lie off it (tips extracted for inclusion below). Includes a nice "procedure" list for tuning apps, and some useful forms for what you should record while tuning. (Page last updated 2000, Added 2000-10-23, Author ?, Publisher HP). Tips:

- Have a performance target.
- Consider architecture and components for bottlenecks.
- Third-party components may have options that cause bottlenecks.
- Having debugging turned on can cause performance problems.
- Having logging turned on can cause performance problems.
- Is the underlying machine powerful enough.
- Carefully document any tests and changes.
- Create a performance baseline.
- Make one change at a time.
- Be careful not to lose a winning tune because it's hidden by a bad tune made at the same time.
- Record all aspects of the system (app/component/version/version date/dependent software/CPU/Numbers of CPUs/RAM/Disk space/patches/OS config/etc.)
- Give the JVMs top system priority.
- Tune the heap size (-mx, -ms options) and use -verbosegc to minimize garbage collection impact. A larger heap reduces the frequency of garbage collection but increases the length of time that any particular garbage collection takes.
- Rules of thumbs are: 50% of free space available after a gc; set the maximum heap size to be 3-4 times the space required for the estimated maximum number of live objects; set the initial heap to size a little below the space required for the average data set, and the maximum value large enough to handle the largest data set; increase -Xmn for applications that create many short-lived objects [is -Xmn a standard option?]. [These rules of thumb should only be considered as starting points. Ultimately you need to tune the VM heap empirically, i.e. by trial and error].
- You may need to add flags to third party products running in the JVM to eliminate explicit calls to garbage collect (VisiBroker has this known problem).
- Watch out for bottlenecks introduced from third party products. Make sure you know and use the options available, many of which can affect performance (for better or worse). Document the changes you make so that you will be able to reproduce the performance.
- computationally intensive applications should increase the number of CPUs to increase overall system performance and throughput.
- Be certain that the application's CPU usage is a factor limiting performance: often, highly contended locks and garbage collections that are too frequent will make the system look busy, but little work is done by the application.

- [Some nice detailed description on how to profile and analyze application problems, from the HP system and JVM level at http://www.devresource.hp.com/JavaATC/JavaPerfTune/symptoms_solutions.html.]

http://www.sys-con.com/java/article.cfm?id=671
J2EE Application server performance (Page last updated April 2001, Added 2001-04-20, Author Misha Davidson, Publisher Java Developers Journal). Tips:

- Good performance has sub-second latency (response time) and hundreds of (e-commerce) transactions per second.
- Avoid n-way database joins: every join has a multiplicative effect on the amount of work the database has to do. The performance degradation may not be noticeable until large datasets are involved.
- Avoid bringing back thousands of rows of data: this can use a disproportionate amount of resources.
- Cache data when reuse is likely.
- Avoid unnecessary object creation.
- Minimize the use of synchronization.
- Avoid using the SingleThreadModel interface for servlets: write thread-safe code instead.
- ServletRequest.getRemoteHost() is very inefficient, and can take seconds to complete the reverse DNS lookup it performs.
- OutputStream can be faster than PrintWriter. JSPs are only generally slower than servlets when returning binary data, since JSPs always use a PrintWriter, whereas servlets can take advantage of a faster OutputStream.
- Excessive use of custom tags may create unnecessary processing overhead.
- Using multiple levels of BodyTags combined with iteration will likely slow down the processing of the page significantly.
- Use optimistic transactions: write to the database while checking that new data is not be overwritten by using WHERE clauses containing the old data. However note that optimistic transactions can lead to worse performance if many transactions fail.
- Use lazy-loading of dependent objects.
- For read-only queries involving large amounts of data, avoid EJB objects and use JavaBeans as an intermediary to access manipulate and store the data for JSP access.
- Use stateless session EJBs to cache and manage infrequently changed data. Update the EJB occasionally.
- Use a dedicated session bean to perform and cache all JNDI lookups in a minimum number of requests.
- Minimize interprocess communication.
- Use clustering (multiple servers) to increase scalability.

http://www.javaworld.com/javaworld/jw-04-2001/jw-0406-syslog.html
Using the Syslog class for logging (Page last updated April 2001, Added 2001-04-20, Author Nate Sammons, Publisher JavaWorld). Tips:

- Use Syslog to log system performance.
- Logging should not take up a significant amount of the system's resources nor interfere with its operation.
- Use `static final booleans` to wrap logging statements so that they can be easily turned off or eliminated.
- Beware of logging to slow external channels. These will slow down logging, and hence the application too.

http://developer.java.sun.com/developer/technicalArticles/Programming/PerfTuning/
Glen McCluskey's article on tuning Java I/O performance. Weak on serialization tuning. (Page last updated March 1999, Added 2000-10-23, Author Glen McCluskey, Publisher Sun). Tips:

- Avoid accessing the disk.
- Avoid accessing the underlying operating system.
- Avoid method calls.
- Avoid processing bytes and characters individually.
- Use buffering either at the class level or at the array level.
- Disable line buffering.
- MessageFormat is slow.
- Reuse objects.
- Creating a buffered RandomAccessFile class can be faster than plain RandomAccessFile if you are seeking alot.
- Compression can help I/O, but only sometimes.
- Use caching to speed I/O.
- Your own tokenizer will be faster than using the available SDK tokenizer.
- Many java.io.File methods are system calls which can be slow.

http://developer.java.sun.com/developer/technicalArticles/ebeans/ejbperformance/
Designing Entity Beans for Improved Performance (Page last updated March 2001, Added 2001-03-21, Author Beth Stearns, Publisher Sun). Tips:

- Remember that every call of an entity bean method is potentially a remote call.
- Designing with one access method per data attribute should only be used where remote access will not occur, i.e. entities are guaranteed to be in the same container.
- Use a value object which encapsulates all of an entity's data attributes, and which transfers all the data in one network transfer. This may result in large objects being transferred though.
- Group entity bean data attributes in subsets, and use multiple value objects to provide remote access to those subsets.

http://www.bastie.de/resource/res/mjp.pdf and
http://www.bastie.de/java/mjperformance/contents.html
Performance tuning report in German. Thanks to Peter Kofler for extracting the tips. (Page last updated November 2001, Added 2001-07-20, Author Sebastian Ritter, Publisher Ritter). Tips:

- Performance optimizations vary in effect on different platforms. Always test for your platforms.
- Reasons not to optimize: can lead to unreadable source code; can cause new errors; optimizations are often compiler/JVM/platform dependent; can lose object orientation.
- Reasons to optimize: application uses too much memory/processor/I/O; application is unnaceptably slow.
- Don't optimize before you have at least a functioning prototype and some identified bottlenecks.
- Try to optimize the design first before targeting the implementation.
- Profile applications. Use the 80/20 rull which suggests that 80% of the work is done in 20% of the code.
- Target loops in particular.
- Monitor running applications to maintain performance.
- Plan and budget for some resources to optimize the application. Try to have or develop a couple of performance experts.

- Specify performance in the project requirements, and specify seperate performance requirements for the various layers of the application.
- Consider the effects of performance at the analysis stage, and include testing of 3rd party tools.
- Use a benchmark harness to make repeatable performance tests, varying the number of users, data, etc. Use profilers and logging to measure performance and identify performance problems.
- Optimize the runtime system if the optimization does not require alterations to the application design or implementation.
- Test various JVMs and choose the optimal JVM.
- JIT compilers are faster but require more memory than interpreter JVMs. HotSpot can provide better performance and a faster startup and maintain a relatively low memory requirement.
- Design in asynchronous operations so tasks are not waiting for others to finish when they don't need to.
- use the right VM
- use the right threading model (native vs. green)
- use native compilers
- give more ram to the VM
- give all ram to short-lived applications to completely avoid GC
- use alternate/optimizing compilers
- use the right database driver
- use direct JDBC drivers
- expand all JDK classes into the filesystem to increase access to classes
- use slot-local variables (1st 128 bit = 4 slots) (applies for interpreters only)
- use int
- use Arraylist instead of Vector
- use own Hashtable implementations for primitives (i.e. int)
- use caches
- use object pools
- avoid remote method calls
- use callbacks to avoid blocking remote method calls
- use batching for remote method calls
- use the flyweight pattern to reduce object creation [The flyweight pattern uses a factory instead of 'new' to reuse objects rather than always create new ones].
- use the right access modifier: static > private > final > protected > public
- use inlining
- use shallow hierarchies (to avoid long instantiation chains)
- use empty default constructors
- use direct variable access (not recommended, breaks OO)
- mix model with view (not recommended, breaks OO)
- use better algorithms
- remove redundant code
- optimize loops
- unroll loops
- use int as loop counter
- count/test loops towards 0
- use Exception terminated loops for long loops
- use constants for expressions with known results, e.g. replace `x = 3; ... (x does not change) ...; x += 3;` with `x = 3; ... (x does not change) ...; x = 6;`
- move code outside loops
- how to optimize: 1st check for better algorithms, 2nd optimize loops

- use shift for *2 and /2
- do not initialize with default values (0, null)
- use char arrays for mutable Strings
- use arrays instead of collections
- use the "private final" modifier
- use System.arraycopy() to copy arrays
- use Hashtable keys with fast hashcode()
- do not use Strings as keys for Hashtables
- use new Hashtable() instaed of Hashtable.clear() for very large Hashtables
- inspect JDK source
- use methods in order: static > final > instance > interface > synchronized
- use own specialized methods instead of JDK's generalized ones
- avoid synchronization
- avoid new objects
- reuse objects
- use the original instead of overloaded constructors (give default parameters by your own)
- avoid inner classes
- use + for concenating 2 Strings, use Stringbuffer for concenating more Strings
- use clone to create new objects (instead of new)
- use instance.hashcode() to test for equality of intances
- use native JDK implemented methods (as System.arraycopy())
- avoid Exceptions (use Exceptions only for cases with probability < 50%, else use error flags)
- combine multiple small try-catchs to one larger block
- use Streams instead of Readers, use Reader and Writer only if you need internationalization
- use buffering for io
- use EOFException and ArrayOutOfBoundsException for terminating io reading loops
- use transient fields to speedup serialisation
- use externalization instead of serialisation
- use multiple threads to increase perceived performance
- use awt instead of swing for speed
- use swing instead of awt for less memory
- use super.paint() to initially draw something (i.e. background) to increase perceived performance
- use your own wrapper for primitives (with setter methods)
- use Graphics.drawPolygon() (native implemented) instead of several Graphics.drawlines().
- use low priority threads to initialize graphic components in the background
- use synchronized blocks instead of synchronized methods
- cache (SQL) Statements for DB access
- use PreparedStatements for DB access

http://java.sun.com/features/2002/03/swinggui.html
Accelerating GUI apps (after 1.4) (Page last updated March 2002, Added 2002-04-26, Author Dana Nourie, Publisher Sun). Tips:

- To add many items to a JComboBox, add them in one go using a Model on a vector, e.g. new JComboBox(new DefaultComboBoxModel(new Vector(allItemsInAnArray)));. This generates only one changed event.
- Perform GUI operations in bulk to minimize the events generated.
- When initializing or totally replacing the contents of a model, construct a new one instead of reusing the existing one to minimize generated events.
- Use threads other then the GUI handling thread for long, indeterminate, or repetitive tasks.

- VolatileImage allows you to create a hardware-accelerated offscreen image and manage the contents of that image.
- From 1.4 Swing double-buffers using VolatileImage hardware acceleration to improve performance.
- Repaint small regions instead of entire sections or screens. For instance, when using tables, repaint a single table cell as needed instead of repainting the entire screen or table.
- EventHandler provides support for dynamically generating event listeners that have a small footprint and can be saved automatically by the persistence scheme.

http://developer.java.sun.com/developer/J2METechTips/2002/tt0325.html
MIDP tips (Page last updated March 2002, Added 2002-04-26, Author Eric Giguere, Publisher Sun). Tips:

- Make HTTP requests in a background thread.
- Use an asynchronous messaging model.
- Use WBXML to compress XML messages.

http://www.javaworld.com/javaworld/jw-09-1996/jw-09-indepth.html
Article about avoiding creating objects where possible. (Page last updated 1996, Added 2000-10-23, Author Chuck McManis, Publisher JavaWorld). Tips:

- "The mythology surrounding the slowness of garbage-collected systems is just that, myth. I can show that the number of instructions executed is the same whether I call malloc() and free() or I only call malloc() and some other code calls free()."
- Simple designs can easily run through many unnecessary objects, e.g. data wrapper objects like Integer.
- Reuse objects where possible.
- Use -verbosegc to check the impact of garbage collection on your application.

http://java.sun.com/people/jag/Fallacies.html
The Eight Fallacies of Distributed Computing (Page last updated 2000, Added 2002-03-25, Author Peter Deutsch, Publisher Sun). Tips:

- The network can fail to deliver at any time.
- Latency is significant.
- Bandwidth is always limited.

http://www.javaworld.com/javaworld/jw-01-2001/jw-0112-performance.html
Article on designing for performance focusing on interfaces (Page last updated January 2001, Added 2001-02-21, Author Brian Goetz, Publisher JavaWorld). Tips:

- Avoid excessive object creation: be wary of object creation inside of tight loops when executing performance-critical code.
- Performance-conscious programmers avoid excessive use of String.
- Defining a utility class which is applied to data required by its constructor means that you must create a new object for every piece of data to run it on. Instead, do not require data in the constructor.
- Do not force methods to provide arguments with input in the form that is convenient rather than efficient. For example, don't require that arguments be passed only as String objects if a byte array or char array would also be functionally equivalent (try to support all formats, especially the efficient ones).

- Defining a method signature in terms of an interchange type (the type of object passed from a caller method to the callee method as an argument) reduces the interface's complexity while maintaining its flexibility, but sometimes this simplicity comes at the cost of performance.

http://java.sun.com/docs/hotspot/PerformanceFAQ.html
HotSpot FAQ (Page last updated August 2000, Added 2001-02-21, Author ?, Publisher Sun). Tips:

- HotSpot has a bunch of startup options that may help you configure your VM to go faster.
- HotSpot garbage collection parameters can be tuned with -Xincgc, -XX:NewSize, -XX:MaxNewSize and -XX:SurvivorRatio(and heap size parameters).
- Sun recommends you no longer use objects pools [this is rather a sweeping and inappropriate statement. Object pools are still useful even with HotSpot, but presumably not as often as previously].
- Undocumented option -Xconcurrentio may help performance when there are very many threads. It uses a lighter thread synchronization model.
- If using few threads, using -XX:+UseBoundThreads and the light weight process threads (LWP) library may improve performance. LWP threads are scheduled by the JVM, system threads have kernel scheduling.
- Don't call System.gc().
- Warming loops is no longer necessary from HotSpot 2.0 (SDK 1.3). HotSpot now supports on-stack-replacement.
- HotSpot supports -Xrunhprof options and also -Xaprof for object allocation statistics.
- Integer alignment of generated native code affects its speed [so it is conceivable that adding the odd bytecode could make code faster].
- HotSpot can eliminate "dead variables" and dead code, i.e. variables that are assigned to but never used [in isolated code segments].
- The generational-GC per object costs varies depending on the length of life of the object.

http://www.unixsolutions.hp.com/products/java/perf.html
A different HP tip page on optimizing Java performance, from the "HP-UX Programmer's Guide for Java". Gives info on HP system performance monitoring too (Page last updated ?, Added 2000-10-23, Author ?, Publisher HP). Tips:

- Maximize thread lifetimes and minimize thread creation/destruction cycles.
- Minimize contention for shared resources.
- Minimize creation of short-lived objects.
- Use -verbosegc to monitor garbage collection. Tune the applications to minimize the effects of garbage collections.
- Disk I/O should be minimized. Don't do random I/O to read a file serially (RandomAccessFile class). You should use buffered I/O.
- Complex AWT graphics will slow down your performance.
- Use the most current version of Java.
- Use -mx and -ms to tune the heap size [now -Xms and -Xmx].
- Profile the code to find bottlenecks.

http://www.artima.com/designtechniques/hotspot.html
Bill Venners on "the right way to optimize" (Page last updated May 1998, Added 2000-10-23, Author Bill Venners, Publisher Artima). Tips:

- Don't optimize until you know you have a problem.
- Measure the program before and after your optimization efforts.

- Profile the program to isolate the code that really matters to performance (10 to 20 percent), and just focus your optimization efforts there.
- Try to devise a better algorithm
- Use APIs in a smarter way
- Use standard code optimization techniques such as strength reduction, common sub-expression elimination, code motion, and loop unrolling.
- Only as a last resort should you sacrifice good object-oriented, thread-safe design and maintainable code in the name of performance.
- Make methods static wherever possible.
- Avoid creating lots of short-lived objects

http://www.informit.com/content/index.asp?product_id={11E331A5-5A08-4FFD-B018-2A7E24D0359B}
Application performance tuning (Page last updated July 2002, Added 2002-07-24, Author Baya Pavliashvili and Kevin Kline, Publisher informIT). Tips:

- Application performance problems can be caused and mitigated with any combination of the following areas: Network topology and throughput; Server hardware configuration; client application code; middle-tier components; database communication code; database configuration settings; logical and physical database design; operating system settings; client hardware; overall application architecture.
- Monitor the application. Primary statistics worth analyzing are: the number of concurrent users; number of transactions per unit of time; duration of the longest and shortest transactions; and the average response time.
- Specify the performance targets.
- Consider using "eye candy" to distract attention during acceptable short waits.
- Identify which application tier contains the bottleneck and fix that. It might be hardware or software; low-level or architecture.
- Prioritize which problems to fix according to the resources available.

http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance.html
Object management article (Page last updated November 1999, Added 2000-12-20, Author Dennis M. Sosnoski, Publisher JavaWorld). Tips:

- Objects have a space overhead in addition to the space taken by the data held by the object.
- Objects have a space overhead in addition to the space taken by the data held by the object. The overhead is dependent on the particular JVM, but there is always some. The space overhead is a per object value, so the percentage of overhead decreases with larger objects. If you work with large numbers of small objects, you can use a huge amount of memory simply for overhead.
- Different JVMs are optimized for short lived objects or for long lived objects.
- Object creation and garbage collection have significant overheads.
- Providing you're sensible about creating objects in heavily used code, it's easy to avoid the object churn cycle.
- The easiest way to reduce object creation in your programs is by using primitive types in place of objects.
- Avoid using wrapper classes (for primitive data types, e.g. Integer) as they impose extra overheads.
- If you're working with a large number of primitive data types, you can avoid the excessive object overhead of wrappers by storing and passing values of the underlying primitive types, and only converting the values into the full objects when necessary for use with methods in the class libraries.

- Avoid convenience classes like Point if you can manage the underlying data directly.
- Reuse objects where possible.
- Use object pools where this is helpful in reusing objects, but be careful that the pool implementation does actually give a performance improvement (dedicated pools within the class can be significantly faster than abstract pool implementations).
- Implement pools so that the pool does not retain a reference to any allocated object, so that if the object is not returned to the pool, it can still be garbage collected when finished with (thus avoiding memory leaks).

http://cin.earthweb.com/public/article/0,,10493_1145241,00.html
Website usability metrics (Page last updated May 2002, Added 2002-07-24, Author Sharon Gaudin, Publisher EarthWeb). Tips:

- A website must be easy to navigate and have a quick display and response time.
- Bad navigation metrics include: abandoned shopping carts; first time visitors look at one or two pages and disappear; dead ends require the "back" button; less than 5% buy something; any broken links.
- Good navigation metrics include: three pages or less from wesbite entry to desired information; no streaming video or Flash introductions; multiple ways to reach the required information; up to date search engines; basic compancy and contact info one click away from the homepage.

http://itmanagement.earthweb.com/ecom/article/0,,11952_1370691,00.html
Common issues affecting Web performance (Page last updated June 2002, Added 2002-07-24, Author Drew Robb, Publisher EarthWeb). Tips:

- Symptoms of network problems include slow response times, excessive database table scans, database deadlocks, pages not available, memory leaks and high CPU usage.
- Causes of performance problems can include the application design, incorrect database tuning, internal and external network bottlenecks, undersized or non-performing hardware or Web and application server configuration errors.
- Root causes of performance problems come equally from four main areas: databases, Web servers, application servers and the network, with each area typically causing about a quarter of the problems.
- The most common database problems are insufficient indexing, fragmented databases, out-of-date statistics and faulty application design. Solutions include tuning the index, compacting the database, updating the database and rewriting the application so that the database server controls the query process.
- The most common network problems are undersized, misconfigured or incompatible routers, switches, firewalls and load balancers, and inadequate bandwidth somewhere along he communication route.
- The most common application server problems are poor cache management, unoptimized database queries, incorrect software configuration and poor concurrent handling of client requests.
- The most common web server problems are poor design algorithms, incorrect configurations, poorly written code, memory problems and overloaded CPUs.
- Having a testing environment that mirrors the expected real-world environment is very important in achieving good performance.
- The deployed system needs to be tested and continually monitored.

http://www.sys-con.com/java/article.cfm?id=1533
The smallest "Hello World" (Page last updated July 2002, Added 2002-07-24, Author Norman Richards, Publisher Java Developers Journal). Tips:

- [Brilliantly amusing search to make the smallest "Hello World" program.]
- Use the -g:none option to strip debugging bytes from classfiles.
- Most bytes in Java class files are from the constant pool, then the method declarations. The constant pool includes class and method names as well as strings.
- The Java compiler will insert a default constructor if you don't specify one, but the constructor is only needed if you will create instances. You can remove the constructor if you will not be creating instances.
- Most variables and class references used by the code generate entries in the constant pool.
- Reusing already existing constant pool entries for class/method/variable names reduces the class file size.

http://www.javaworld.com/javaworld/jw-11-2000/jw-1110-smartproxy.html
Article on using smart proxies. (Page last updated November 2000, Added 2001-01-19, Author M. Jeff Wilson, Publisher JavaWorld). Tips:

- Use smart proxies to transparently cache data in the client, thus reducing the number of remote calls.
- Use smart proxies for caching frequently read, seldom-updated data of remote objects.
- Use smart proxies to monitor the performance of RMI calls.
- Use smart proxies to prevent returning multiple copies of the same remote object to client code.

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf
Paper detailing the "Best Practices for Developing High Performance Web and Enterprise Applications" using IBM's WebSphere. All the tips are generally applicable to servlet/EJB development, as well as other types of server development. (Page last updated September 2000, Added 2001-01-19, Author Harvey W. Gunther, Publisher IBM). Tips:

- Do not store large object graphs in javax.servlet.http.HttpSession. Servlets may need to serialize and deserialize HttpSession objects for persistent sessions, and making them large produces a large serialization overhead.
- Use the tag "<%@ page session="false"%>" to avoid creating HttpSessions in JSPs.
- Minimize synchronization in Servlets to avoid multiple execution threads becoming effectively single-threaded.
- Do not use javax.servlet.SingleThreadModel.
- Use JDBC connection pooling, release JDBC resources when done, and reuse datasources for JDBC connections.
- Use the HttpServlet Init method to perform expensive operations that need only be done once.
- Minimize use of System.out.println.
- Avoid String concatenation "+=".
- Access entity beans from session beans, not from client or servlet code.
- Reuse EJB homes.
- Use Read-Only methods where appropriate in entity-beans to avoid unnecessary invocations to store.
- Use the lowest impact transaction level possible for each transaction.

- The EJB "remote programming" model always assumes EJB calls are remote, even where this is not so. Where calls are actually local to the same JVM, try to use calling mechanisms that avoid the remote call.
- Remove stateful session beans (and any other unneeded objects) when finished with, to avoid extra overheads in case the container needs to be passivated.
- Beans.instantiate() incurs a filesystem check to create new bean instances. Use "new" to avoid this overhead.

http://www-4.ibm.com/software/webservers/appserv/3steps_perf_tuning.pdf
Tuning IBM's WebSphere product. White paper: "Methodology for Production Performance Tuning". Only non-product specific Java tips have been extracted here. (Page last updated September 2000, Added 2001-01-19, Author Gennaro (Jerry) Cuomo, Publisher IBM). Tips:

- A size restricted queue (closed queue) allows system resources to be more tightly managed than an open queue.
- The network provides a front-end queue. A server should be configured to use the network queue as its bottleneck, i.e. only accept a request from the network when there are sufficient resources to process the request. This reduces the load on an app server. However, sufficient requests should be accepted to ensure that the app server is working at maximum capacity, i.e. try not to let a component sit idle while there are still requests that can be accepted even if other components are fully worked.
- Try to balance the workload of the various components.
- [Paper shows a nice throughput curve giving recommended scaling behavior for an server]
- The desirable target bottleneck is the CPU, i.e. a server should be tuned until the CPU is the remaining bottleneck. Adding CPUs is a simple remedy to this.
- Use connection pools and cached prepared statements for database access.
- Object memory management is particularly important for server applications. Typically garbage collection could take between 5% and 20% of the server execution time. Garbage collection statistics provide a useful monitor to determine the server's "health". Use the verbosegc flag to collect basic GC statistics.
- GC statistcs to monitor are: total time spent in GC (target less than 15% of execution time); average time per GC; average memory collected per GC; average objects collected per GC.
- For long lived server processes it is particularly important to eliminate memory leaks (references retained to objects and never released).
- Use -ms and -mx to tune the JVM heap. Bigger means more space but GC takes longer. Use the GC statistics to determine the optimal setting, i.e the setting which provides the minimum average overhead from GC.
- The ability to reload classes is typically achieved by testing a filesystem timestamp. This check should be done at set intermediate periods, and not on every request as the filesystem check is an expensive operation.

http://www.redbooks.ibm.com/abstracts/sg245657.html
WebSphere V3 Performance Tuning Guide (Page last updated March 2000, Added 2001-01-19, Authors Ken Ueno, Tom Alcott, Jeff Carlson, Andrew Dunshea, Hajo Kitzhöfer, Yuko Hayakawa, Frank Mogus, Colin D. Wordsworth, Publisher IBM). Tips:

- [The Red book lists and discusses tuning parameters available to Websphere]
- Run an application server and any database servers on separate server machines.
- JVM heap size: -mx, -ms [-Xmx, -Xms]. As a starting point for a server based on a single JVM, consider setting the maximum heap size to 1/4 the total physical memory on the server and setting the minimum to 1/2 of the maximum heap. Sun recommends that ms be set to somewhere between 1/10 and 1/4 of the mx setting. They do not recommend setting ms and

mx to be the same. Bigger is not always better for heap size. In general increasing the size of the Java heap improves throughput to the point where the heap no longer resides in physical memory. Once the heap begins swapping to disk, Java performance drastically suffers. Therefore, the mx heap setting should be set small enough to contain the heap within physical memory. Also, large heaps can take several seconds to fill up, so garbage collection occurs less frequently which means that pause times due to GC will increase. Use verbosegc to help determine the optimum size that minimizes overall GC.

- In some cases turning off asynchronous garbage collection ("-noasyncgc", not always available to all JVMs) can improve performance.
- Setting the JVM stack and native thread stack size (-oss and -ss) too large (e.g. greater than 2MB) can significantly degrade performance.
- When security is enabled (e.g. SSL, password authentication, security contexts and access lists, encryption, etc) performance is degraded by significant amounts.
- One of the most time-consuming procedures of a database application is establishing a connection to the database. Use connection pooling to minimize this overhead.

http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-ternary.html
Using a ternary search tree for fast searches of partial text matches (Page last updated February 2001, Added 2001-03-21, Author Wally Flint, Publisher JavaWorld). Tips:

- [Article discusses several efficient algorthms for searching through ternary search trees which provide fast partial match searches of character array keys].

http://www-106.ibm.com/developerworks/java/library/j-threads1.html
When synchronization is required (Page last updated July 2001, Added 2001-07-20, Author Brian Goetz, Publisher IBM). Tips:

- synchronization means mutual exclusion (if the same monitor is used), atomicity of the synchronized block (again with respect to other threads using the same monitor) and synchronization of thread memory to main memory.
- Because synchronization synchronizes thread memory with main memory, there is a cost to synchronization beyond simply acquiring a lock.
- Too little synchronization can lead to corrupt data; too much can lead to reduced performance and deadlock.
- The costs of synchronization vary with JVMs, with more recent JVMs being more efficient.
- The costs of synchronization differs depending on whether or not threads are actually contending for locks (more expensive, slower), or for uncontended synchronization where the thread is basically acting in single-threaded mode (cheaper, faster).
- You need to synchronize or make `volatile` variables holding data that will be shared between threads.
- Composite operations may need synchronizing to make them atomic even if each individual operation is already synchronized.

http://www-106.ibm.com/developerworks/java/library/j-threads2.html
Reducing thread contention (Page last updated September 2001, Added 2001-10-22, Author Brian Goetz, Publisher IBM). Tips:

- Thread contention impairs scalability because it forces the scheduler to serialize operations, even if a free processor is available.
- Analyze your program to determine where contention is likely to occur.
- Make synchronized blocks as short as possible.
- Spread synchronizations over more than one lock.

- [Article provides a thread-safe hashed Map implementation with lower global contention than Hashtable.]
- If you will be acquiring and releasing the same lock many times (such as in a loop), acquire the lock before the loop: it is faster to acquire a lock that you already hold than one that nobody holds.

http://www.onjava.com/pub/a/onjava/2002/04/03/javaenterprise_tips.html
J2EE worst practices (Page last updated April 2002, Added 2002-04-26, Author Brett McLaughlin, Publisher OnJava). Tips:

- The choice of data store type (RDB, ODB, XML-DB, directory-server, etc) affects performance, and should not be made without performance considerations.
- Directory servers are optimized for frequent reads, with few writes. If you frequently add data to a directory server, performance degrades.
- Stateless session beans are soooo much faster.

http://www.javaworld.com/javaworld/jw-12-2001/jw-1207-hprof.html
The hprof profiler (Page last updated December 2001, Added 2001-12-26, Author Bill Pierce, Publisher JavaWorld). Tips:

- Use the hprof profiler with the startup command "java -Xrunhprof[:help][:<suboption>=<value>,...] MyMainClass".
- [Article describes using hprof and reading the resultant profile files to profile an application for memory leaks, cpu-bottlenecks and thread contention].
- hprof can be used to profile object allocation (heap option), method bottlnecks (cpu option) and thread contention (monitor option).

http://www.weblogic.com/docs51/admindocs/tuning.html
Weblogic tuning (generally applicable Java tips extracted) (Page last updated June 2000, Added 2001-03-21, Author BEA Systems, Publisher BEA). Tips:

- Response time is affected by: contention and wait times, particularly for shared resources; and software and hardware component performance, i.e. the amount of time that resources are needed.
- A well-designed application can increase performance by simply adding more resources (for instance, an extra server).
- Use clustered or multi-processing machines; use a JIT-enabled JVM; use Java 2 rather than JDK 1.1;
- Use -noclassgc. Use the maximum possible heap size that also is small enough to avoid the JVM from swapping (e.g. 80% of RAM left over after other required processes). Consider starting with minimum initial heap size so that the garbage collector doesn't suddenly encounter a full heap with lots of garbage. Benchmarkers sometimes like to set the heap as high as possible to completely avoid GC for the duration of the benchmark.
- Distributing the application over several server JVMs means that GC impact will be spread in time, i.e. the various JVMs will most likely GC at different times from each.
- On Java 1.1 the most effective heap size is that which limits the longest GC incurred pause to the longest acceptable pause in processing time. This will typically require a *reduction* in the maximum heap size.
- Too many threads causes too much context switching. Too few threads may underutilize the system. If n=number of threads, k=number of CPUs, then: (n < k) results in an under utilized CPU; (n == k) is theoretically ideal, but each CPU will probably be under utilized; (n > k) by a "moderate amount of threads" is practically ideal; (n > k) by "many threads" can lead to

significant performance degradation from context switching. Blocked threads count for less in the previous formulae.
- Symptoms of too few threads: CPU is waiting to do work, but there is work that could be done; Can not get 100% CPU; All threads are blocked [on i/o] and runnable when you do an execution snapshot.
- Symptoms of too many threads: An execution snapshot shows that there is a lot of context switching going on in your JVM; Your performance increases as you decrease the number of threads.
- If many client connections are dropped or refused, the TCP listen queue may be too short.
- Try to avoid excessive cycling (creation/deletion or activation/passivation) of beans.

http://www.weblogic.com/docs51/techdeploy/jdbcperf.html
Weblogic JDBC tuning (Page last updated April 1999, Added 2001-03-21, Author BEA Systems, Publisher BEA). Tips:

- Use connection pools to the database and reuse connections rather than repeatedly opening and closing connections. Optimal pool size is when the connection pool is just large enough to service requests without waits.
- Cache frequently requested data in the JVM and avoid the unnecessary database requests.
- Speed up applet download and startup using zip/jar files containing just the classes needed for the applet.
- Avoid accessing the database wherever possible.
- Fetch rows in batches rather than one at a time, using the batch as a read-ahead mechanism (i.e. pre-fetch rows in batches). Tune the batch size and the number of rows pre-fetched. Avoid pre-fetching BLOBs.
- Avoid moving data unless absolutely necessary. Process the data and produce results as close to its source as possible. Use stored procedures.
- Streamline data before the result crosses the network.
- Use stored procedures to avoid extra network transfers.
- Use built-in DBMS set-based processing to operate on multiple rows/tables in one request.
- Avoid row at a time processing, process multiple rows together wherever possible.
- Counting entries in a table (e.g. using `SELECT count(*) from myTable, yourTable where ...`) is resource intensive. Try first selecting into temporary tables, returning only the count, and then sending a refined second query to return only a subset of the rows in the temporary table.
- Proper use of SQL can reduce resource requirements. Use queries which return the minimum of data needed: avoid `SELECT *` queries. A complex query that returns a small subset of data is more efficient than a simple query that returns more data than is needed.
- Make your queries as smart as possible, i.e. as precise as possible to minimize the data transferred to just that subset that is required.
- Try to batch updates: collect statements together and execute them together in one transaction. Use conditional logic and temporary variables if necessary to achieve statement batching.
- Never let a DBMS transaction span user input.
- Consider using optimistic locking. Optimistic locking employs timestamps to verify that data has not been changed by another user, otherwise the transaction fails.
- Use in-place updates, i.e. change data in rows/tables that already exist rather than adding or deleting rows/tables. Try to avoid moving rows or changing their sizes.
- Store operational data and historic data separately (or more generally store frequently used data separately from infrequently used data).
- Keep your operational data set as small as possible, to avoid having to read through data that is irrelevant.

- DBMSs work well with parallelism. Try to design the application to do other things while interacting with the DBMS.
- Use pipelining and parallelism. Designing applications to support lots of parallel processes working on easily distinguished subsets of the work makes the application faster. If there are multiple steps to processing, try to design your application so that subsequent steps can start working on the portion of data that any prior process has finished, instead of having to wait until the prior process is complete.
- Choose the right driver for your application, i.e. the fastest JDBC driver.

http://www.sys-con.com/websphere/article.cfm?id=40
JDBC optimizing for DB2 (Page last updated April 2002, Added 2002-04-26, Author John Goodson, Publisher WebSphere Developers Journal). Tips:

- Use the same connection to execute multiple statements.
- Keep connection objects open, and reuse them, rather than repeatedly connecting and disconnecting.
- Turn off autocommit, but don't leave transactions open for too long.
- Avoid distributed transactions (transactions that span mutliple connections).
- Minimize the data retrieved from the database, both columns and rows. Use setMaxRows, setMaxFieldSize, and SetFetchSize.
- Use the most efficiently handled data type: character strings are faster than integers, which are in turn more efficient than floating-point and timestamps.
- Use programmatic updates: updateXXX() calls on updatable resultsets. The resultset is already postioned at a row, so eliminating the usual overhead of finding the row to be updated when using an UPDATE statement.
- Cache any required metadata and use metadata methods as rarely as possible as they are quite slow.
- Avoid using null parameters in metadata queries.
- Use a dummy query to get the metadata for a column, rather than use the getcolumns()
- Use parameter markers with stored procedures, rather than embedding data literally in the statement, to minimize parsing overheads.
- Use prepared statements for repeatedly executing SQL statements
- Choose the optimal cursor: forward-only for sequential reads; insensitive for two-way scrolling. Avoid insenstive cursors for queries that only return one row.

http://www.sys-con.com/java/article.cfm?id=1171
J2EE Performance tuning (Page last updated October 2001, Added 2001-10-22, Author James McGovern, Publisher Java Developers Journal). Tips:

- Call HttpSession.invalidate() to clean up a session when you no longer need to use it.
- For Web pages that don't require session tracking, save resources by turning off automatic session creation using: <%@ page session="false"%>
- Implement the HttpSessionBindingListener for all beans that are scoped as session interface and explicitly release resources implementing the method valueUnbound().
- Timeout sessions more quickly by setting the timeout or using session.setMaxInactiveInterval().
- Keep-Alive may be extra overhead for dynamic sites.
- Use the include directive <%@ include file="copyleft.html" %> where possible, as this is a compile-time directive (include action <jsp:include page="copyleft.jsp" /> is a runtime directive).
- Use cache tagging where possible.
- Always access entity beans from session beans.

- If only using an entity bean for data access, use JDBC directly instead.
- Use read-only in the deployment descriptor.
- Cache access to EJB homes.
- Use local entity beans when beans are co-located in the same JVM.
- Proprietary stubs can be used for caching and batching data.
- Use a dedicated remote object to generate unique primary keys.
- Follow standard JDBC optimizations: use connection pools; prefer stored procedures or direct SQL; use type 4 drivers; remove extra columns from the result set; use prepared statements when practical; have your DBA tune the query; choose the appropriate transaction levels.
- Consider storing all database character data in Unicode to eliminate conversion overheads. But beware: this step will cause your database size to grow, as Unicode requires 2 bytes per character.
- Use block fetches when the query will give a large ResultSet and all rows are needed. Use the Page-by-Page Iterator pattern when only some of the rows may be needed.
- Consider using an in-memory database (product) for data that doesn't need to be persisted.
- Use an algorithm to prune caches to stop them growing too large.
- Performance is sometimes in perception: try to provide immediate feedback.
- Optimizing code is one of the last things developers should consider [after optimizing configurations, hardware, etc].

http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-merlin.html
Using nonblocking I/O and memory-mapped buffers in SDK 1.4. (Page last updated September 2001, Added 2001-10-22, Author Michael T. Nygard, Publisher JavaWorld). Tips:

- Before SDK 1.4, servers had a number of performance problems: i/o could easily be blocked; garbage was easily generated when reading i/o; many threads are needed to scale the server.
- Many threads each blocked on i/o is an inefficient architecture in comparison to one thread blocked on many i/o calls (multiplexed i/o).
- Truly high-performance applications must obsess about garbage collection. The more garbage generated, the lower the application throughput.
- A Buffer (java.nio.*Buffer) is a reusable portion of memory. A MappedByteBuffer can map a portion of a file directly into memory.
- Direct Buffer objects can be read/written directly from Channels, but nondirect Buffer objects have a data copy performed for read/writes to i/o (and so are slower and may generate garbage). Convert nondirect Buffers to direct Buffers if they will be used more than once.
- Scatter/gather operations allow i/o to operate to and from several Buffers in one operation, for increased efficiency. Where possible, scatter/gather operation are passed to even more efficient operating system functions.
- Channels can be configured to operate blocking or non-blocking i/o.
- Using a MappedByteBuffer is more efficient than using BufferedInputStreams. The operating system can page into memory more efficiently than BufferedInputStream can do a block read.
- Use Selectors to multiplex i/o and avoid having to block multiple threads waiting on i/o.

http://www.sys-con.com/java/article.cfm?id=1408
Combining apps in one JVM (Page last updated April 2002, Added 2002-04-26, Author Kirk Pepperdine, Publisher Java Developers Journal). Tips:

- Loading multiple applications in the same JVM allows resource sharing and reduce system memory requirements.
- Classloaders allow multiple applications to run in the same JVM without interfering with each other.
- [Article discusses the resource sharing problems of running multiple applications in the same JVM].

http://portals.devx.com/datadirect/Article/6338
JDBC Drivers (Page last updated March 2002, Added 2002-04-26, Author Barrie Sosinsky, Publisher DevX). Tips:

- Type 1 drivers are JDBC-ODBC bridges, plus an ODBC driver. Recommended only for prototyping, not for production. Not suitable for high-transaction environments. Not well supported, and limited in functionality.
- Type 2 drivers use a native API, and are part-Java drivers. Have a binary-code client loading overhead, and may not be fully-featured.
- Type 3 drivers are a pure Java driver which connects to database middleware. Can be server-based which is frequently faster than types 1 and 2.
- Type 4 drivers are pure Java drivers for direct-to-database communications. This can minimize overheads, and generally provides the fastest driver.
- JDBC 3.0 has additional features to improve performance such as advancements in connection pooling, statement pooling, RowSet objects.
- Opening a connection is the most resource-expensive step in database transactions. Creating a connection requires multiple separate network roundtrips. However, once the connection object has been created, there is little penalty in leaving the connection object in place and reusing it for future connections.
- Connection pooling, keeps open a cache of database connection objects, making them available for immediate use. Instead of performing expensive network roundtrips to the database server to open a connection, a connection attempt results in the re-assignment of a connection from the local cache.
- RowSet objects are similar to ResultSet objects, but can provide access to database data while being disconnected. This allows data to be efficiently cached in its simplest form.
- Prepared statement pooling (available from JDBC 3.0) caches SQL queries that have been previously optimized and run so that, should they be needed again, they do not have to go through optimization pre-processing again (avoiding optimization steps, such as checking syntax, validating addresses, and optimizing access paths and execution plans). Statement pooling can be a significant performance booster.
- Statement pooling and connection pooling in JDBC 3.0 can cooperate to share statement pools, so that connections that can use a cached statement from another connection, thus incurring statement preparation overheads only once on the first execution of some SQL by any connection.
- Database drivers developed by vendors other than the the database vendor can be better performing and more feature full. (Driver vendors concentrate on the driver, database vendors have many other things to consider).
- Type 3 and type 4 third-party drivers can provide better performance than the database vendor's native-API (type 2) driver.
- Try to use a driver that supports JDBC 3.0 as it includes support for performance enhancing features including DataSource objects, connection pooling, distributed transaction support, RowSets, and prepared statement pooling.
- Type 3 and Type 4 drivers are the drivers to use when performance is important.

http://developer.java.sun.com/developer/Books/EarlyJ2SE/IO.pdf
Shortened version of chapter 2, "I/O", from "Early Adopter J2SE 1.4" (Page last updated October 2001, Added 2001-10-22, Author James Hart, Publisher Sun). Tips:

- Non-blocking I/O can improve performance by minimizing the amount of time spent in I/O calls, though they may add complexity to the application.
- The old I/O classes can now be interrupted more reliably from 1.4.
- FileChannel.transferFrom() is an efficient way to copy data between files.

http://developer.java.sun.com/developer/Books/EarlyJ2SE/Using.pdf
Shortened version of chapter 5, "Utilities: The Logging Architecture", from "Early Adopter J2SE 1.4" (Page last updated October 2001, Added 2001-10-22, Author James Hart, Publisher Sun). Tips:

- Logging can take place asynchronously: a call to log can return before the log has been formatted and written.
- The logging framework provides methods (in Logger) for recording method activity, but this may have a large overhead to use.

http://www.AmbySoft.com/javaCodingStandards.pdf
Coding standards with a small but interesting section (section 7.3) on optimizations (Page last updated January 2000, Added 2001-04-20, Author Scott Ambler, Publisher AmbySoft). Tips:

- Optimizing code is one of the last things that programmers should be thinking about, not one of the first.
- Don't optimize code that already runs fast enough.
- Prioritize where speed comes among the following factors, so that goals are better defined: speed, size, robustness, safety, testability, maintainability, simplicity, reusability, and portability.
- The most important factors in looking for code to optimize are fixed overhead and performance on large inputs: fixed overhead dominates speed for small inputs and the algorithm dominates for large inputs (a program that works well for both small and large inputs will likely work well for medium-sized inputs).
- Operations that take a particular amount of time, such as the way that memory and buffers are handled, often show substantial time variations between platforms.
- Users are sensitive to particular delays: users will likely be happier with a screen that draws itself immediately and then takes eight seconds to load data than with a screen that draws itself after taking five seconds to load data.
- Give users immediate feedback: you do not always need to make your code run faster to optimize it in the eyes of your users.
- Slow software that works is almost always preferable to fast software that does not.

http://win-www.uia.ac.be/~s985218/professional/thesis/archief/documenten/Marktoverzicht.doc
Overview of common application servers. (Announced at http://www.theserverside.com/home/thread.jsp?thread_id=9581). I've extracted the performance related features (Page last updated October 2001, Added 2001-10-22, Author Pieter Van Gorp, Publisher Van Gorp). Tips:

- Load balancing: random; minimum load; round-robin; weighted round-robin; performance-based; load-based; dynamic algorithm based; dynamic registration.
- Clustering. Additionally: distributed transaction management; in-memory replication of session state information; no single point of failure.
- Connection pooling.

- Caching. JNDI caching. Distributed caching with synchronization.
- Thread pooling.
- Configurable user Quality of Service.
- Analysis tools.
- Low system/memory requirements.
- Optimized subsystems (RMI, JMS, JDBC drivers, JSP tags & cacheable page fragments).
- Optimistic transaction support.

http://www.onjava.com/pub/a/onjava/2001/11/07/atomic.html
Atomic File Transactions. (Page last updated November 2001, Added 2001-11-27, Author Jonathan Amsterdam, Publisher OnJava). Tips:

- If you don't require powerful search capabilities, using flat files may be faster than dealing with a database.
- Basic file operations (deletion, creation, renaming) are atomic. Other operations and combinations of operations are not atomic. Atomicity can be built but comes at a performance cost. You will have to determine whether the increase in robustness is worth the slowdown in your application.
- Do the I/O in a background thread to mitigate the performance impact of adding atomicity to file transactions.
- [Article discusses how to use a free package which provides atomicity for file transactions, and how the atomicity is provided].

http://www.onjava.com/pub/a/onjava/2002/02/06/atomic.html
Atomic File Transactions, Part 2 (Page last updated February 2002, Added 2002-02-22, Author Jonathan Amsterdam, Publisher OnJava). Tips:

- [Article continues implementation of a framework for atomic file transactions].
- If a transaction creates a file and then performs several other actions on it, there is no need to undo the actions -- it is enough to delete the file.
- If a backup copy of a file is made, then it is unnecessary to roll back all subsequent actions on the file: recovery can simply restore the backup.

http://wireless.java.sun.com/midp/ttips/memory/
MIDP memory tuning (Page last updated June 2002, Added 2002-07-24, Author Jonathan Knudsen, Publisher Sun). Tips:

- Use an obfuscator to minimize the size of classes.
- Minimize resource sizes by using as few images as possible, and using fewer colors in the images you do use.
- Use as few objects as possible.
- Dereference objects (set them to null) when they're no longer useful so they will be garbage-collected.
- Catch OutOfMemoryErrors on all allocations, or at least the large ones. Don't let an OutOfMemoryError take your application by surprise.
- MIDlets use three types of memory: program memory, heap, and persistent storage. Each of these may be scarce and they should all be treated with respect.

http://java.sun.com/blueprints/patterns/j2ee_patterns/catalog.html
Design patterns catalog (Page last updated 2001, Added 2002-01-25, Author ?, Publisher Sun). Tips:

- [Page lists some patterns with summaries and links to detailed info. Patterns are: Data Access Object; Fast-Lane Reader; Front Controller; Page-by-Page Iterator; Session Facade; Value Object].
- Use the Data Access Object pattern to decouple business logic from data access logic, allowing for optimizations to be made in how data is managed.
- Use the Fast-Lane Reader pattern to accelerate read-only data access by not using enterprise beans.
- Use the Front Controller pattern to centralize incoming client requests, allowing optimizations to be made in aggregating the resulting view.
- Use the Page-by-Page Iterator pattern to efficiently access a large, remote list by retrieving its elements one sublist of value objects at a time.
- Use the Session Facade pattern to provide a unified, workflow-oriented interface to a set of enterprise beans, thus minimizing client calls to server EJBs.
- Use the Value Object pattern to efficiently transfer remote, fine-grained data by sending a coarse-grained view of the data.

http://www.sys-con.com/java/article.cfm?id=1268
EJB design (Page last updated January 2002, Added 2002-01-25, Author Boris Lublinsky, Publisher Java Developers Journal). Tips:

- Some application server implementations (e.g., WebSphere) automatically convert remote communications to local communications to make them faster.
- Low granularity (i.e. fine-grained) methods in an EJB typically leads to poor performance of the overall system.
- Local interfaces in EJB 2.0 is one attempt to improve overall performance: local interfaces provide for beans in the same container to interact locally without involving RMI.
- The most effective way to improve the overall performance of EJB-based applications is to minimize the amount of method invocations, making the communications overhead negligible compared with the execution time. This can be achieved by implementing coarse-grained methods.
- Entity beans should not be simply mapped to database tables. Treating entity beans as such fine-grained objects which are effectively wrappers on table rows leads to increased network communications and heavier database communications than if entity beans are treated as coarse-grained components.
- For optimal performance, entity beans should be designed to: have large granularity, which usually means they should contain multiple Java classes and support multiple database tables; be associated with a certain amount of persistent data, typically multiple database tables, one of which should define the primary key for the whole bean; support meaningful business methods and encapsulate business rules to access the data.
- Don't use client transactions in the EJB environment since long-running transactions that can cause database lockup.
- Entity beans are transactional resources due to their stateful nature, but application server vendors often rely on the underlying database to lock and resolve access appropriately. Although this approach greatly improves performance, it provides the potential for database lockup.

http://developer.java.sun.com/developer/technicalArticles/J2EE/despat/
Design Patterns (Page last updated January 2002, Added 2002-01-25, Author Vijay Ramachandran, Publisher Sun). Tips:

- [Article discusses several design patterns: Model-View-Controller, Front Controller, Session Facade, Data Access Object].

- Use the Front Controller pattern to channel all client requests through a single decision point, which allows the application to be balanced at runtime.
- Use a Session Facade to provide a simple interface to a complex subsystem of enterprise beans, and to reduce network communication requirements.
- Use Data Access Objects to decouple the business logic from the data access logic, allowing data access optimizations to be decoupled from other types of optimizations.

http://www.onjava.com/pub/a/onjava/2002/01/16/patterns.html
J2EE Design Patterns for the presentation tier (Page last updated January 2002, Added 2002-01-25, Author Sue Spielman, Publisher OnJava). Tips:

- [Article discusses several design patterns: Intercepting Filter, Front Controller, View Helper, Composite View, Service To Worker, Dispatch View. Performance is not explicitly covered, but at least a couple are relevant to getting good performance].

http://www.devx.com/upload/free/Features/Javapro/2002/02feb02/kr0202/kr0202-1.asp
Thread programming (Page last updated January 2002, Added 2002-01-25, Author Karthik Rangaraju, Publisher DevX). Tips:

- Use Djikstra semaphores (synchronized acquire()/release()) to control access to a finite pool of resources.
- Conditional events provide a more sophisticated version of the wait()/notify() mechanism which avoids some potential problems of that mechanism.
- Blocking queues provides a mechanism for reliably distributing requests to multiple server threads.
- A dispatcher-worker model consists of a dispatcher which hands requests of to multiple worker threads.
- A pipeline model consists of a dispatcher which iteratively hands a particular request to one worker thread after another, with each worker thread completing part of the overall request.

http://portals.devx.com/Intel/Article/6441
Some (Intel chip) optimization myths debunked. (Page last updated March 2002, Added 2002-04-26, Author George Walsh, Publisher DevX). Tips:

- If optimization and performance tools are used throughout development rather than tacked on at the end as a final "optimization phase," time to market and costs can actually be decreased by speeding up the process of locating problems and bottlenecks in code.
- Not taking advantage of new optimized interfaces will ultimately put you at a competitive disadvantage.

http://www.sys-con.com/java/article.cfm?id=1280
Benchmarking (Page last updated January 2002, Added 2002-01-25, Author Glenn Coates & Carl Barratt, Publisher Java Developers Journal). Tips:

- A benchmark should exhaustively exercise the platform being tested and produce a numerical result corresponding to the speed of the benchmark.
- Benchmarks can be simplistic, made up of simple routines executed successively. Each routine should run for a reasonable amount of time.
- Ensure that performance statistics are not lost within start-up overheads.
- Don't rely on standard third-party benchmarks, as these may not apply to your application characteristics.

- Benchmarks could include looking at the speed of code execution; the response time of the user interface; implementation of the garbage collector; and memory issues.
- In J2ME, software accelerators may improve speed, but they may also consume excessive power compared to a processor that executes Java as its native language.
- No benchmark can replace the actual user application. At the earliest possible stage in the design process, application developers must run their own code on the proposed hardware.

http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-dcl.html
Double-checked locking revisited. (Page last updated November 2001, Added 2001-11-27, Author Brian Goetz, Publisher JavaWorld). Tips:

- Double-checked locking is not guaranteed to produce consistent results.
- Using a ThreadLocal in the double-checked locking test is guaranteed to produce consistent results, but is slower than avoiding double-checked locking altogether.
- ThreadLocal is faster in each SDK release through 1.2, 1.3 and 1.4. 1.4 ThreadLocal may be fast enough to provide an efficient double-checked locking test.

http://www.onjava.com/pub/a/onjava/2001/10/17/rmi.html
Command objects for RMI. (Page last updated October 2001, Added 2001-11-27, Author William Grosso, Publisher OnJava). Tips:

- Use Command objects to automatically queue or retry RMI calls.

http://www.onjava.com/pub/a/onjava/2001/10/31/rmi.html
Caching RMI stubs. (Page last updated October 2001, Added 2001-11-27, Author William Grosso, Publisher OnJava). Tips:

- Remote method calls are much slower than local calls, at least 1000 times slower.
- Reduce the number of remote calls made by an application to improve performance.
- Cache remote objects locally where possible, rather than repeatedly fetching them.
- Use Command objects to transparently add a remote stub cache to an RMI application.
- Caching stubs keeps them from being garbage collected, and may prevent an RMI server from closing. Use a policy to expire stubs and delete them from the cache.

http://intranetjournal.com/articles/200110/gb_10_24_01a.html
Website performance. (Page last updated October 2001, Added 2001-11-27, Author Gordon Benett, Publisher Intranet Journal). Tips:

- Some e-commerce consultants cite an attention span on the order of eight seconds as the threshold for abandoning a slow retail site.
- Where broadband connections are the norm, pages that don't appear instantly stand a good chance of never being seen: slow pages might as well be no pages.
- Systems can only be designed to meet performance goals if those goals have been identified. Determine what range of response times will be acceptable.
- Try to understand the performance impacts of your design decisions. However the performance of some design choices can be hard to predict and may remain unclear before testing.
- Test the system under conditions that simulate real patterns of use.
- Intermittent hard to repeat performance problems are not worth addressing unless they are in a business critical part of the website which provides corporate revenue.
- Use a rapid, iterative development process in combination with frequent performance testing.

- Try to plan up-front rather than have to rely on late-phase tuning.

http://java.sun.com/products/java-media/2D/perf_graphics.html
High performance graphics (Page last updated February 2002, Added 2002-03-25, Author ?, Publisher Sun). Tips:

- The large number extra features and increased cross-platform compatibility added to the Java Graphics framework in SDK 1.2 made the graphics slower than the 1.1 Graphics. SDK 1.4 targeted these performance issues head on.
- VolatileImage allows you to create hardware-accelerated offscreen images, resulting in better performance of Swing and gaming applications in particular and faster offscreen rendering.
- When filling a shape with a complex paint, Java 2D must query the Paint object every time it needs to assign a color to a pixel whereas a simple color fill only requires iterating through the pixels and assigning the same color to all of them.
- The graphics pipeline (from SDK 1.4) only gets invalidated when an attribute is changed to a different type of value, rather than when an attribute is changed to a different value of the same type. For example rendering one opaque color is the same rendering another opaque color, so this would not invalidate the pipeline. But changing an opaque color to a transparent color would invalidate the pipeline.
- Smaller font is rendered faster than larger font.
- Hardware-accelerated scaling is currently (1.4.0 release) disabled on Win32 because of quality problems, but you can enable it with a runtime flag, -Dsun.java2d.ddscale=true.
- From SDK 1.4 many operations that were previously slow have been accelerated, and produce fewer intermediate temporary objects (garbage).
- Alpha blending and anti aliasing adversely affect performance.
- Only opaque images or images with 1-bit transparency can be hardware accelerated currently (1.4.0).
- Use 1-bit transparency to make the background color of a sprite rectangle transparent so that the character rendered in the sprite appears to move through the landscape of your game, rather than within the sprite box.
- Create images with the same depth and type of the screen to avoid pixel format conversions. Use either Component.createImage() or GraphicsConfiguration.createCompatibleImage(), or use a BufferedImage created with the ColorModel of the screen.
- Rectangular fills--including horizontal and vertical lines--tend to perform better than arbitrary or non-rectangular shapes whether they are rendered in software or with hardware acceleration.
- If your application must repeatedly render non-rectangular shapes, draw the shapes into 1-bit transparency images and copy the images as needed.
- If you experience low frame rates, try commenting out pieces of your code to find the particular operations that are causing problems, and replace these problem operations with something that might perform better.
- Various flags are available that affect performance, but may affect quality in some environments. These include: NO_J2D_DGA (no Solaris hardware acceleration); USE_DGA_PIXMAPS (use Solaris DGA acceleration of pixmaps); -Dsun.java2d.noddraw=true (turn off DirectDraw); -Dsun.java2d.ddoffscreen=false (disable DirectDraw offscreen acceleration); -Dsun.java2d.ddscale=true (enable hardware acceleration in Win32); -Dsun.java2d.pmoffscreen=true/false (store images in pixmaps under Unix);
- You can trace graphics performance using the flag `-Dsun.java2d.trace=<optionname>,<optionname>,...` where the options are `log` (print

primitives on execution); `timestamp` (timestamp log entries); `count` (print total calls of each primitive used); `out:<filename>` (send logs to filename); `verbose` (whatever); `help` (help);

http://developer.java.sun.com/developer/JDCTechTips/2002/tt0409.html
Assertions (Page last updated April 2002, Added 2002-04-26, Author Glen McCluskey, Publisher Sun). Tips:

- Disabled assertions add a cost of one check of a global state flag
- Enabled assertions add a cost of a check of a global state flag and evaluating the boolean expression. Also the cost of throwing a new exception is added if the assertion fails.
- Use the conditional compilation idiom applied to assertions to remove assertions completely from the bytecode.

http://dcb.sun.com/practices/devnotebook/gc_perspective.jsp
GC performance tuning (Page last updated February 2002, Added 2002-03-25, Author Alka Gupta and Michael Doyle, Publisher Sun). Tips:

- The point when garbage collection kicks in is out of the control of the application. This can cause a sequential overhead on the application, as the garbage collector suspends all application threads when it runs, causing inconsistent and unacceptable application pauses, leading to high latency and decreased application efficiency.
- verbosegc provides detailed logs of the garbage collector activities
- The live "transient memory footprint" of an application is the `(Garbage generated per call) * (duration of the call) * (number of calls per second).`
- GC pause time caused by two-space collection of short-lived objects is directly proportional to the size of the memory space allocated to holding short-lived objects. But smaller available space can mean more frequent GCs.
- Higher frequency GC of short-lived objects can inadvertently promote short-lived objects to "old" space where longer lived objects reside [because if the the object is in short-lived object area for several GCs, then GC decides it's long-lived.] The promoteAll option will force the GC to assume that any object surviving GC of young space is long-lived, and is immediately promoted to old space..
- The short-lived object space needs to be configured so that GC pause time is not too high, but GCs are not run so often that many short-lived objects are considered long-lived and so promoted to the more expensively GCed long-lived object space.
- The long-lived object space needs to be large enough to avoid an out-of-memory error, but not so high that a full GC of old space pauses the JVM for too long.
- [Article covers 1.2 and 1.3 GC memory space models].
- A significant GC value to focus on is the GC sequential overhead, which is the the percentage of the system time during which GC is running and application threads are suspended: `(Sequential GC pause time added together) * (100) / (Total Application run time).`
- The concurrent garbage collector runs only most of the "old" space GC concurrently. Some of the "old" space GC and all the "young" space GC is sequential.
- GC activity can take hours to settle down to its final pattern. Fragmentation of old space can cause GC times to degrade, and it may take a long time for the old space to become sufficiently fragmented to show this behavior.
- GC options can reduce fragmentation (such as bestFitFirst).
- The promoteAll option produced a significant improvement in performance [which I find curious].

http://www.sys-con.com/java/article.cfm?id=1412
Mobile & wireless devices (Page last updated April 2002, Added 2002-04-26, Author James White, Publisher Java Developers Journal). Tips:

- Prototype to determine the performance of your device. Wireless transmissions require testing to determine if the transfer rates and processing times are acceptable.
- Attempt to create applications that can accomplish 80% or more of their operations through the touch of a single key/button or the "tap" or touch of the stylus to the screen.
- Trying to manipulate a very small scroll bar on a small screen can be an exercise in hand-eye coordination. Horizontal scrolling should be avoided at all costs. Use "jump-to" buttons rather than scrollbars.
- Try to avoid having the user remember any data, or worse, having to compare data across screens.
- Performance will always be a concern in J2ME.
- Avoid garbage generation: Use StringBuffer for mutable strings; Pool reusable instances of objects like DateFormat; Use System.gc() to jump-start or push the garbage collection process.
- Compile the code with debugging information turned off using the -g:none switch. This increases performance and reduces its footprint.
- Avoid deep hierarchies in your class structure.
- Consider third-party JVMs, many are faster than the Sun ones.
- Small XML parsers and micro databases are available for purchase where necessary.

http://developer.java.sun.com/developer/J2METechTips/2002/tt0226.html
Minimizing bytecode size for J2ME (Page last updated February 2002, Added 2002-03-25, Author Eric Giguere, Publisher Sun). Tips:

- Eliminate unnecessary features.
- Avoid inner classes: make the main class implement the required Listener interfaces and handle the callbacks there.
- Use built-in classes if functionality is close enough, and work around their limitations.
- Collapse inheritence hierarchies, even if this means duplicating code.
- Shorten all names (packages, classes, methods, data variables). Some obfuscators can do this automatically. MIDP applications are completely self-contained, so you can use the default package with no possible name-clash.
- Convert array initialization from code to extract data from a binary string or data file. Array initialization generates many bytecodes as each element is separately initialized.

http://developer.java.sun.com/developer/technicalArticles/Programming/JVMPerf/
Sun engineering report on performance tests of various configurations of the 1.2.2 and 1.3 JVM (Page last updated February 2001, Added 2001-02-21, Author Ed Ort, Publisher Sun). Tips:

- Different versions of the Sun JVM support different optimization flags. Some flags may allow you to configure the garbage collector generational spaces.
- Configure heap space using -Xms and -Xmx [-ms and -mx for 1.1.x JVMs] to optimize the JVM heap memory for improved performance.
- If the JVM supports configuring the garbage collector generational spaces (-Xgenconfig in 1.2.2; -XX:newSize, -XX:MaxNewSize, -XX:SurvivorRatio in 1.3), then you can improve performance by specifying generation spaces more appropriate for your application [you can start with some appropriate configuration depending on the ratios of short-lived to medium-lived to long-lived objects, then test multiple configurations to determine the optimal config].

- The 1.3 JVM appears to be faster when run with the -server flag.
- The -Xoptimize flag seems to improve performance on those 1.2.x JVMs that support it.

http://www.jguru.com/jguru/faq/view.jsp?EID=131579
Discussion on JDBC performance (Page last updated August 2000, Added 2001-02-21, Author , Publisher JGuru). Tips:

- Use a connection pool mechanism whenever possible.
- Use prepared statements.
- Use stored procedures.
- Select only required columns rather than using select * from Table xyz.
- Always close Statement and ResultSet objects as soon as possible.
- Work with DatabaseMetaData to get information about database functionality.
- Always catch *and* handle database warnings and exceptions.
- Time DB queries.
- Use the most appropriate datatype specific kinds of data, e.g. store dates as a date type rather than varchar.
- Use scrollable ResultSet (JDBC 2.0).
- Stay away from the JDBC-ODBC and other Type 1 drivers where possible.

http://www.theserverside.com/resources/article.jsp?l=J2EEPerformance
Improving J2EE performance (Page last updated May 2002, Added 2002-07-24, Author Scott Marlow, Publisher The Server Side). Tips:

- Set performance goals before development starts.
- If supporting clients with slow connections, consider compressing data for network communication.
- Minimize the number of network round trips required by the application.
- For applications to scale to many users, minimize the amount of shared memory that requires updating.
- Cache data to minimize lookup time, though this can reduce scalability if locks are required to access the cache.
- If there are more accesses than updates to a cache, share the access lock amongst all the accessors, though be aware that this reduces the window for updators to lock the cache.
- For optimum performance, zero shared memory provides a cache per user.
- Be methodical to ensure that changes for performance do actually improve performance.
- Eliminate memory leaks before tuning execution speed.
- Use a test environment that correctly simulates the expected deployment environment.
- Simulate the expected client activity, and compare the performance against your expected goals.
- Consider which metrics to measure, such as: Max response time under heavy load; CPU utilization under heavy load; How the application scales as additional users are added.
- Profile the application to find the bottlenecks. Correct bottlenecks by making one change at a time and testing for improvement.
- Generate stack traces to look for bottlenecks which are multi-thread conflicts (waiting for locks).
- Improving the performance of a method that is called 1000 times and takes a tenth of a second on average each call, is better than improving the performance of a method that is only called 10 times but takes 1 second each call.
- Don?t cache data unless you know how and when to invalidate the cached entries.

http://jsl.jcon.org/javaperformance.html
An assortment of tips (Page last updated 2000, Added 2000-10-23, Author Curt Smith, Publisher Smith). Tips:

- Use the Java compiler?s optimization flag (javac -O)
- Profile the application (using -prof) & re-code the methods that are taking the longest.
- Avoid repeatedly instantiating exceptions. Reuse exceptions in preference.
- Move common subexpressions to one execution.
- Eliminate casts, or reduce the number of casts being made.
- Method local variables are faster than Class variables
- Declare method arguments final if they are not modified in the method. In general declare all variables final if they are not modified after being initialized or set to some value.
- Declare methods private and/or final whenever that makes sense. This can help the compiler inline methods. [final methods are of dubious value]
- Buffer i/o. Use BufferedReaders.
- DON?T create static strings via new().
- Use String.intern() to reduce the number of strings in your runtime. [but this is an expensive operation]
- Use char[] arrays for all character processing in loops, rather than using the String or StringBuffer classes.
- StringBuffer default size is 16 chars. Set this to the maximum expected string length.
- StringTokenizer is inefficient. It can be optimized by storing the string and delimiter in a character array instead of in a String, or by storing the highest delimiter character to allow a quicker check.
- Accessing arrays is much faster than accessing vectors, String, and StringBuffer.
- Use System.arraycopy() to improve performance.
- Initialize expensive arrays in class static initializers, and create a per instance copy of this array initialized with System.arrarycopy().
- Vector is convenient to use, but inefficient. For best performance, use it only when the structure size is unknown, and efficiency is not a concern.
- When using Vector, ensure that elementAt() is not used inside a loop.
- Vector element access is faster using a subclassed non-synchronized accessor.
- Re-use Vectors by using Vector.removeAllElements().
- Initialize Vector to the maximum expected size.
- Re-use Hashtables by using Hashtable.clear().
- Set the Hashtable size to be large enough to hold the expected elements. Use a prime number for table size.
- Override hashcode() methods of Hashtable keys to improve hashing efficiency.
- Use non-synchronized hash table classes.
- Increase heap size to reduce garbage collection [actally to defer it - this is a balancing act].
- Use the -verbosegc option to monitor garbage collection.
- Use arrays of smaller datatypes (short rather than int) is possible.
- Avoid allocating objects in loops (readLine() is a common example).
- Minimizing synchronization may take work, but can pay off well.
- Polling is only acceptable when waiting for outside events and should be performed in a "side" thread. Use wait/notify instead.
- Eliminate calls to synchronized methods (but be careful of being overly ambitious in this).
- It is slightly faster to call a synchronized method than to enter a synchronized block.
- Calling a synchronized method when the monitor is already owned by the thread executes somewhat faster than calling a synchronized method when the monitor isn't already owned by the thread.
- Creating objects is expensive.

- Consider reusing objects in reuse pools.
- Move new(), invariants and constant conditionals outside of loops.
- Unroll loops.
- Make tests in loops as simple as possible.
- Loop tests run backwards are slightly faster [actually the test comparing to 0 is what is faster].
- Use local variables, rather than any other type of variable, in loops.
- Combine similar loops. Nest the busiest loop, if loops are interchangeable.
- Convert expressions to table lookups [doesn't always work].
- Cache values that are expensive to fetch or compute.
- Pre-compute results.
- Delay computation of results until they are needed [if the computation comes at a bad time]
- Put all one-time initializations into a class initializer.

http://www.nandighosha.org/forum/topic.asp?TOPIC_ID=185&FORUM_ID=10&CAT_ID=2&Topic_Title=Java+performance+tuning+tips&Forum_Title=Java+%2D+Tips+Of+The+Day
Various performance tips (Page last updated May 2001, Added 2001-06-18, Author Asha Balasubramanyan, Publisher Nandighosha). Tips:

- Use buffered I/O. Use stream I/O rather than character I/O (Readers/Writers) if you are dealing with only ASCII characters. Avoid premature flushing of buffers.
- Recycle objects. try to minimize the number of objects you create in your java programs.
- Factor out constant computations from loops. Push one-time computations into methods called once only.
- Use StringBuffer when dealing with mutable strings. Initialize the StringBuffer with the proper size.
- Comparison of two string objects is faster if they differ in length.
- Avoid converting Strings to bytes and back.
- StringTokenizer is slow. Write your own tokenizer.
- Use charAt() instead of StartsWith() in case you are looking for a single character within a String.
- Avoid premature object creation. Creation should be as close to the actual place of use as possible.
- Avoid initializing twice.
- Zeroing buffer contents is not usually required.
- Be careful about the order of evaluation of expressions with OR and AND conditions.
- Use ArrayList for non-synchronized Vectors.
- Minimize JNI calls in your code.
- Minimize calls to Date and related classes.

http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-deadlock.html
Avoiding synchronization deadlocks (Page last updated October 2001, Added 2001-10-22, Author Brain Goetz, Publisher JavaWorld). Tips:

- Deadlocks are difficult to identify from code analysis, and can occur unexpectedly.
- Always acquire locks in the same order to avoid one common cause of deadlocking. If you can guarantee that all locks will always be acquired in a consistent order, then your program will not deadlock.
- Try to avoid acquiring more than one lock at a time (though this is usually impractical).
- Keep synchronized blocks of code as short as possible.

http://www.javaspecialists.co.za/archive/Issue038a.html
Counting object creation (Page last updated December 2001, Added 2002-02-22, Author Heinz M. Kabutz, Publisher Kabutz). Tips:

- Add a counter in to the Object constructor to trace object creation. Doesn't trace arrays [nor objects created from deserialization].

http://www.sys-con.com/java/article.cfm?id=1149
Performance tuning (Page last updated September 2001, Added 2001-10-22, Author James McGovern, Publisher Java Developers Journal). Tips:

- Often there's a trade-off between designing for reuse and designing for performance. Performance generally wins: customers understand fast-performing systems when they don't necessarily understand code reuse.
- Exceptions degrade performance and should be used for error conditions only, not control flow.
- Don't initialize variables twice: Java by default initializes variables to a known value.
- Use the factory pattern to enable reuse or cloning of objects.
- Make classes final.
- Use local variables as much as possible.
- Use non-blocking I/O (available from 1.4, or use www.cs.berkeley.edu/~mdw/proj/java-nbio/download.html for earlier versions).
- Create/Use method interfaces that reduce overhead.
- Use bit-shifting instead of multiplication or division by powers of two.
- Choose the JVM that runs your application fastest.
- Use clustering application servers.
- Avoid stateful sessions.
- Profile and tune the application (architecture and code).
- Set aside at least 20% of the total project time for performance.
- Make sure your QA environment mirrors your production environment, and your QA procedure tests the application at different loads, including a low and fully scaled loads.

http://www.sys-con.com/weblogic/article.cfm?id=58
Why CMP is better than BMP (Page last updated April 2002, Added 2002-04-26, Author Tyler Jewell, Publisher Weblogic Developers Journal). Tips:

- Use CMP except in specific cases when BMP is necessary: fields use stored procedures; persistence is not simple JDBC (e.g. JDO); One bean maps to multiple tables; non-standard SQL is used.
- CMP can make many optimizations: optimal locking; optimistic transactions; efficient lazy loading; efficiently combining multiple queries to the same table (i.e. multiple beans of the same type can be handled together); optimized multi-row deletion to handle deletion of beans and their dependents.

http://www.theserverside.com/resources/article.jsp?l=Building-Sclable-Recoverable-Applications
Scalable recoverable applications (Page last updated May 2002, Added 2002-07-24, Author Billy Newport, Publisher The Server Side). Tips:

- [Article describes several approaches to building a scalable recoverable system]
- Split the application into a transactional part and a non-transactional part. The non-transactional part can be replicated.

- Using a single machine limits both reliability and scalability. Scalability is completely dependent on how powerful the single machine can become.
- Multiple front-end machines with http request load balancing is more reliable, but the database machine is still a single point of failure.
- A database caching layer in the servlet helps performance. An EJB caching layer is difficult to achieve.
- Oracle 9i includes queryable snapshots of the main database which can offload the query to run against the clients local snapshot.
- An in-memory database (such as TimesTen) is very, very fast and can act as a queryable cache for a back end database.
- Database instances on each machine, with replication increases reliability and access speed. But updates now need to be handled differently. Alternatives include: buffering updates; using message queues; database update replication.
- Partitioning the database across multiple machines adds scalability, but must be done with care.
- If you want very reliable systems then everything has to be controlled.
- A load balancing message queue may be needed for a high rate of messages (>500/sec).
- Note that reliable systems should ensure that all duplicated data have no single points of failure in the software or hardware chain behind the data (different controllers, UPSs, etc).

http://www.sys-con.com/java/article.cfm?id=1133
Techniques to avoid deadlocks (Page last updated September 2001, Added 2001-10-22, Author Mark Dykstra, Publisher Java Developers Journal). Tips:

- Potential deadlocks can be caused by coding styles.
- Always acquire a set of locks in the same set order.
- Don't hold a lock and wait for an event.
- Specify which thread should have access to data at any time.
- Ensure that both access and update to the same variable is synchronized on the same monitor.

http://www.onjava.com/pub/a/onjava/excerpt/bldgjavaent_8/index3.html
Stateful to Stateless Bean (Page last updated February 2002, Added 2002-03-25, Author Brett McLaughlin, Publisher OnJava). Tips:

- Stateless session beans are much more efficient than stateful session beans.
- Stateless session bean have no state. Most containers have pools of stateless beans. Each stateless bean instance can serve multiplw clients, so the bean pool can be kept small, and doesn't need to change in size avoiding the main pooling overheads.
- A separate stateful bean instance must exist for every client, making bean pools larger and more variable in size.
- [Article discusses how to move a stateful bean implementation to stateless bean implementtaion].

http://www.ddj.com/documents/ddj0204a/
Alternatives to using 'new'. (Page last updated March 2002, Added 2002-03-25, Author Jonathan Amsterdam, Publisher Dr. Dobb's). Tips:

- The 'new' operator is not object oriented, and prevents proper polymorphic object creation.
- Constructors must be made non-public and preferably private to limit the number of objects of a class.

- The Singleton pattern and the Flyweight (object factory) pattern are useful to limit numbers of objects of various types and to assist with object reuse and reduce garbage collection.
- The real-time specification for Java allows 'new' to allocate objects in a 'current memory region', which may be other than the heap. Each such region is a type of MemoryArea, which can manage allocation.
- Using variables to provide access to limited numbers of objects is efficient, but a maintenance problem if you need to change the object access pattern, for example from a global singleton to a ThreadLocal Singleton.
- A non-static factory method is polymorphic and so provides many advantages over static factory methods.
- The Abstract Factory design pattern uses a single class to create more than one kind of object.
- An alternative to the Flyweight pattern is the Prototype pattern, which allows polymorphic copies of existing objects. The Object.clone() method signature provides support for the Prototype pattern.
- Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. Then you could keep already-initialized objects in a table, and clone an existing object instead of expensively creating a new one from scratch.
- Immutable objects can be returned directly when using Prototyping, avoiding the copying overhead.

http://www.javacoffeebreak.com/articles/network_timeouts/index.html
Timing out sockets (Page last updated 2000, Added 2001-06-18, Author David Reilly, Publisher JavaCoffeeBreak). Tips:

- Use a timer thread to monitor socket activity and timeout if blocked.
- Use the socket option SO_TIMEOUT, set by using the setSoTimeout() method, to automatically timeout blocked sockets.

http://www.javaspecialists.co.za/archive/Issue001.html
Deadlocks (Page last updated November 2000, Added 2002-04-26, Author Heinz M. Kabutz, Publisher Kabutz). Tips:

- Use CTRL+BREAK to get a thread dump when a deadlock occurs, to find where the deadlock is.
- Use SwingUtlities.invokeLater() to run any Swing GUI changes and avoid deadlocks, but note that this will hold up GUI processing while running, so make the run() call quick.
- Use SwingUtilities.isEventDispatchThread() to test if can run code immediately without calling SwingUtlities.invokeLater().

http://www.ibm.com/developerworks/java/library/j-load
Load testing of web applications (Page last updated June 2001, Added 2001-06-18, Author Frank Cohen, Publisher IBM). Tips:

- Current Web-application architectures consists many small servers that are accessed through a load balancer, providing a front-end to a powerful database server. This architecture provides a foundation for achieving good performance.
- Load testing of web applications should include: State machine testing (entries in a shopping basket, should still be there when checked out); Really long session testing (session started then continued several hours later); Hordes of savage users testing (users do lots nonsensical activity); Privileged testing (only some users should be able to access some functionality);

Speed testing (do tasks complete within the required times?). Each type of test should be run with several different user loads.
- Test suites should be automated and easily changed.
- [Article discusses *Load*, an open-source set of tools with XML scripting language]

http://www.javaworld.com/javaworld/javaone01/j1-01-patterns.html
J2EE design patterns to improve performance (Page last updated June 2001, Added 2001-06-18, Author Daniel H. Steinberg, Publisher JavaWorld). Tips:

- Combine multiple remote calls for state information into one call using a value object to wrap the data (the Value Object pattern, superceded by local interfaces in EJB 2.0).
- Where long lists of data are returned by queries, use the Page-by-Page Iterator pattern: a server-side object that holds data on the server and supplies batches of results to the client.

http://www.onjava.com/pub/a/onjava/2001/12/19/oraclejdbc.html
Oracle JDBC tips (Page last updated December 2001, Added 2001-12-26, Author Donald Bales, Publisher OnJava). Tips:

- Although Oracle recommend using the OCI driver for optimal client side access, the writer finds the Thin driver to have have better performance.
- Turn off autocommit, Connection.setAutoCommit(false).
- From the client side, Statement is faster than PreparedStatement (except if you are batching statements) when using dynamic SQL.
- Use PreparedStatements for all, except dynamic, SQL statements.
- Use PreparedStatements for batching repetitive inserts or updates.
- OraclePreparedStatement.setExecuteBatch() (proprietary method) is the fastest way to execute batch statements.
- Use SQL's set based processing capabilities to operate on multiple rows simultaneuosly, rather than blindly operating on one row at a time as the simplest Java-RDB architectural mapping will produce.

http://www.oreilly.com/catalog/jorajdbc/chapter/ch19.html
Chapter 19, "Performance" of Java Programming with Oracle JDBC (Page last updated December 2001, Added 2001-12-26, Author Donald Bales, Publisher O'Reilly). Tips:

- Performance should be considered at the start of a project.
- Use the EXPLAIN PLAN facility to explain how the database's optimizer plans to execute your SQL statements, to identify performance improvements such as additional indexes.
- If more than one SQL statement is executed by your program, you can gain a small performance increase by turning off auto-commit.
- It takes about 65 iterations of a prepared statement before its total time for execution catches up with a statement, because of prepared statement initialization overheads.
- Use PreparedStatements to batch statements for optimal performance.
- The Thin driver is faster than the OCI driver. This is contrary to Oracle's recommendation.
- A SELECT statement makes two round trips to the database, the first for metadata, the second for data. Use OracleStatement.defineColumnType() to predefine the SELECT statement, thus providing the JDBC driver with the column metadata which then doesn't require the first database trip.
- Given a simple SQL statement and a stored procedure call that accomplishes the same task, the simple SQL statement will always execute faster because the stored procedure executes the same SQL statement but also has the overhead of the procedure call itself. On the other

hand complex tasks requiring several SQL statements can be faster using stored procedures as fewer network trips and data transfers will be needed.

http://www.fawcette.com/javapro/2002_01/magazine/columns/weblication/
Database performance (Page last updated December 2001, Added 2001-12-26, Author Peter Varhol, Publisher JavaPro). Tips:

- Thoughtful page design makes for a better user experience by enabling the application to seem faster than it really is.
- Use the flush method associated with the out object to display static text and graphics on the browser page before the database query returns, to prevent the user from having to look at a blank page for a long time.
- ResultSet types affect updates. TYPE_FORWARD_ONLY: no updating allowed; TYPE_SCROLL-SENSITIVE: update immediately; TYPE_SCROLL_INSENSITIVE: update when the connection is closed. (Concurrency type must be set to CONCUR-UPDATABLE to allow the table to be updated.)
- Performance can be better if changes to the database are batched: turn off autocommit; add multiple SQL statements using the Statement.addBatch() method; execute Statement.executeBatch().
- Scaled systems need optimized SQL calls, querying the right amount of data, and displaying pages before the query is complete.
- Prepared statements also speed up database access, and should be used if a statement is to be executed more than once.

http://www-105.ibm.com/developerworks/education.nsf/java-onlinecourse-bytitle/56A6275393F0BE5786256AFD004DBBB4?OpenDocument
JDBC tutorial (requires free registration) (Page last updated November 2001, Added 2001-12-26, Author Robert J. Brunner, Publisher IBM). Tips:

- Type 1 (JDBC-ODBC-DB) drivers incur a performance penalty because of the bridging needed to reach the database.
- [Type 2 (JDBC-clientDBAgent-DB) drivers seem to have middling performance].
- Type 3 (JDBC-Middleware-DB) drivers incur a performance penalty because of the bridging needed to reach the database, but does introduce optimization potential from the location of the middleware.
- Type 4 (JDBC-DB) drivers typically provide optimum driver performance.
- The higher the level of transaction protection, the higher the performance penalty. Transaction levels in order of increasing level are: TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITTED, TRANSACTION_READ_COMMITTED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE. Use Connection.setTransactionIsolation() to set the desired tansaction level.
- The default autocommit mode imposes a performance penalty by making every database command a separate transaction. Turn off autocommit (Connection.setAutoCommit(false)), and explicitly specify transactions.
- Batch operations by combining them in one transaction, and in one statement using Statement.addBatch() and Statement.executeBatch().
- Savepoints (from JDBC3.0) require expensive resources. Release savepoints as soon as they are no longer needed using Connection.releaseSavepoint().
- Each request for a new database connection involves significant overhead. This can impact performance if obtaining new connections occurs frequently. Reuse connections from connection pools to limit the cost of creating connections. [The tutorial lists all the overheads involved in creating a database connection].

- The ConnectionPoolDataSource (from JDBC3.0) and PooledConnection interfaces provide built-in support for connection pools.
- Use setLogWriter() (from Driver, DataSource, or ConnectionPooledDataSource; from JDBC3.0) to help trace JDBC flow.
- Use Connection.setReadOnly(true) to optimize read-only database interactions.
- Use Connection.nativeSQL() to see how the SQL query will execute in the database to help ensure that the SQL is optimized.

http://www-105.ibm.com/developerworks/education.nsf/java-onlinecourse-bytitle/975BFD2C367CFFD686256B0500581B3B?OpenDocument
Advanced JDBC tutorial (requires free registration). (Page last updated November 2001, Added 2001-12-26, Author Robert J. Brunner, Publisher IBM). Tips:

- PreparedStatement objects are compiled (prepared) by the JDBC driver or database for faster performance, and accept input parameters so they can be reused with different data.
- Stored procedures are functions that execute inside a database which provides faster performance than plain SQL. Java supports stored procedures from CallableStatement objects.

http://developer.java.sun.com/developer/technicalArticles/J2EE/J2EEpatterns/
Performance optimizing design patterns for J2EE (Page last updated December 2001, Added 2001-12-26, Author Vijay Ramachandran, Publisher Sun). Tips:

- For read-only access to a set of data that does not change rapidly, use the Fast Lane Reader pattern which bypasses the EJBs and uses a (possibly non-transactional) data access object which encapsulates access to the data. Use the Fast Lane Reader to read data from the server and display all of them in one shot.
- When you need to access a large remote list of objects, use the Page-by-Page Iterator pattern which sends smaller subsets of the data as requested until the client no longer want any more data. Use the Page-by-Page Iterator to send lists of simple objects from EJBs to clients.
- When the client would request many small data items which would require many remote calls to satisfy, combine the multiple calls into one call which results in a single Value Object which holds all the data required to be transferred. Use the Value Object to send a single coarse-grained object from the server to the client(s).

http://developer.java.sun.com/developer/J2METechTips/2001/tt0725.html
Flicker-free graphics with the Mobile Information Device Profile (Page last updated July 2001, Added 2001-08-20, Author Eric Giguere, Publisher Sun). Tips:

- Use double buffering: draw into an offscreen buffer, then copy into the display buffer. Copying buffers is very fast on most devices, while directly drawing to a display sometimes causes users to see a flicker, as individual parts of the display are updated. Double buffering avoids flickering by combining multiple individual drawing operations into a single copy operation.
- Use the Canvas.isDoubleBuffered() method, to determine if double buffering is already automatically used: on some implementations the Canvas object's paint method is already a Graphics object of an offscreen buffer managed by the system. (The system then takes care of copying the offscreen buffer to the display.)
- Use javax.microedition.lcdui.Image class to create an offscreen memory buffer, and use Graphics to draw to the offscreen buffer and to copy the contents of the offscreen buffer onto the display. The offscreen buffer is created by calling one of the Image.createImage methods.

- Double buffering does have some overhead: if only making small changes to the display, it might be slower to use double buffering.
- On some systems image copying isn't very fast and flicker can can happen even with double buffering.
- Keep the number of offscreen buffers to a minimum. There is a memory penalty to pay for double buffering: the offscreen memory buffer can consume a large amount of memory.
- Free the offscreen buffer whenever the canvas is hidden (use the canvas' hideNotify() and showNotify() methods.)

http://www.ibiblio.org/javafaq/quotes1999.html
Some killer quotes, leading to the odd tip. (Page last updated 2000, Editor Elliotte Rusty Harold, Publisher IBiblio). Tips:

- A Vector may be convenient and generalized, but it's almost always overkill, and you pay the price for it in speed and other ways. --*Greg Guerin on the MRJ-dev mailing list*
- A lot of speed (or memory) can go down the drain if the underlying structure is a poor fit to the problem, or is inefficient for a particular program's common actions. --*Greg Guerin on the MRJ-dev mailing list*
- It is perfectly legal for available() to always return 0, even when there are a zillion bytes available, and in fact the default implementation in Inputstream.available() does just that. --*Thomas Maslen on the mrj-dev mailing list*
- Seeing the wrong solution to a problem (and understanding why it is wrong) is often as informative as seeing the correct solution. --*W. Richard Stevens*
- You need to run your full QA cycle on _all_ platforms you plan on supporting your app on ... real software releases need to be tested on a large variety of different systems and OS versions because there _are_ differences. Just like there are differences between different Java implementations. --*Jens Alfke on the mrj-dev mailing list*
- I often find with Java that if you run the same program twice, the second run is significantly faster, presumably because the JVM is remembering something. --*Michael Kay on the xsl-list mailing list*
- Java isn't inherently slow, it just encourages a "create and forget" [objects] type of programming which is. --*Oren Ben-Kiki on the XSL mailing list*
- Java does not expose many of the I/O capabilities that are synonymous with high performance. Examples include memory mapped files and asynchronous I/O. Heck, it doesn't even expose non-blocking I/O. --*Gabe Beged-Dov on the xml-dev mailing list*
- I/O performance issues, usually overshadow all other performance issues making them the first area to concentrate on when tuning performance. Unfortunately, optimal reading and writing can be challenging in Java. --*Daniel Lord and Achut Reddy, http://www.sun.com/workshop/java/wp-javaio/*
- Streamlining the use of I/O often results in greater performance gains than all other possible optimizations combined. --*Daniel Lord and Achut Reddy http://www.sun.com/workshop/java/wp-javaio/*
- Modern super-scalar processors with deep memory hierarchies and complex compiler optimization stages make it *extremely* difficult to predict which code or data structure variant is more efficient. Old rules of thumb and "common sense" are not of much use any more for distinguishing more and less performant algorithms of comparable complexity on a late 1990s processor. Surprises are frequent. Design decisions on performance grounds should today only be made after real measurements and much of what you learned 10 years ago about manual optimization is obsolete these days. --*Markus Kuhn on the Unicode mailing list*

- Most Java VM implementations search the interface list back to front so that most often used interface should be the last interface in the 'implements' list. *--Don Park on the xml-dev mailing list*

http://www.javaworld.com/javaworld/jw-11-2000/jw-1117-performance.html
Article about optimizing I/O performance. (Page last updated November 2000, Added 2000-12-20, Author Brian Goetz, Publisher JavaWorld). Tips:

- Measure early, measure often. You can't effectively manage performance if you don't know the source of your problem.
- Spending days tuning a subsystem that accounts for 1 percent of an application's total runtime simply cannot yield more than a 1 percent improvement in application performance.
- Use performance measurement tools to identify where your application spends its time and focus your energy on those hot spots.
- Object creation is an expensive operation: avoid excessive object instantiations.
- Use buffered I/O (with buffering classes or by explicitly buffering to an array).
- InputStream runs faster than Reader.
- Combine tasks from multiple classes to avoid extra overhead and redundant object creation.

http://www.theparticle.com/javadata2.html
Particle's pretty good coverage of the main Java data structures. Only a few tuning tips: reuse, pools, optimized sorting. But knowing which structure to use for a particular problem is an important performance tuning technique. (Page last updated April 2000, Added 2000-12-20, Author J. Particle, Publisher Particle). Tips:

- Make linked lists faster by having dummy first and last nodes.
- Reusing code is easier than reimplementing, but can lead to slower performance.
- Use node pools to reduce memory impact.
- Sorting elements on insertion means they don't need to be sorted later.
- [Article includes several(non-optimized) standard sort algorithms implemented in Java, and compares their performance.]
- [Article discusses optimizing a quicksort.]
- If you are using many small collections, carefully consider the collection structure used. Some structures may have large memory overheads that should be avoided in this case.
- Some discussion of hidden surface removal for graphics.

http://www.javaworld.com/javaworld/javatips/jw-javatip78.html
Article on recycling resource pools (Page last updated 1998, Added 2000-12-20, Authors Philip Bishop and Nigel Warren, Publisher JavaWorld). Tips:

- Check for broken resources when putting them back in the pool.
- Use the builder pattern: break the construction of complex objects into a series simpler Builder objects, and a Director object which combines the Builders to form the complex object. Then you can use Recycler (a type of Director) to replace only the broken parts of the complex object, so reducing the amount of objects that need to be recreated.

http://www.javaworld.com/jw-06-1998/jw-06-object-pool.html
Article on building an object pool for improved performance. (Page last updated June 1998, Added 2000-12-20, Author Thomas E. Davis, Publisher JavaWorld). Tips:

- [Article discusses generic pool issues including storage, tracking, and expiration times of pool elements.]

- Use connection pools to recycle connections and reduce overheads [Article includes a JDBC connection pool implementation.]

http://www.javaworld.com/jw-08-1998/jw-08-object-pool.html
Article on improving object pools performance. (Page last updated September 1998, Added 2000-12-20, Author Thomas E. Davis, Publisher JavaWorld). Tips:

- Use an expiration thread to clean up excessive amounts of objects in the pool.
- Use java.lang.ref.Reference objects to determine when objects checked out but never checkd in have been released by the application.
- Limiting the size of the pool can adversely impact performance.

http://www.sys-con.com/java/article.cfm?id=725
Optimizing JDBC (Page last updated August 2001, Added 2001-08-20, Author John Goodson, Publisher Java Developers Journal). Tips:

- Minimize the use of Metadata: Cache all metadata as they will not change; Avoid using null arguments in metadata methods; Use a dummy query with getMetadata() rather than getColumns().
- Retrieve data as efficiently as possible: Minimize the amount of data returned by the query; Don't make average users pay the same query cost of the users with extensive query requirements; Remember that users seldom want to see too much data in one go; Use setMaxRows(), setMaxFieldSize(), and SetFetchSize(); Decrease the column size; Use the smallest packet size that will meet your needs (if the driver supports packet sizing).
- Use a parametrized remote procedure call (RPC) rather than passing parameters as part of the RPC call, e.g. use `Connection.prepareCall("Call getCustName (?)").setLong (1,12345)` rather than `Connection.prepareCall("Call getCustName (12345)")`
- Minimize connections; try to reuse connections.
- Turn autocommit off.
- Avoid using distributed transactions.
- Use getBestRowIndentifier() to determine the optimal set of columns to use in the *Where* clause for updating data. (The columns returned could be pseudo-columns that can provide pointers to the exact location of the data, and are not obtained by getColumns().)

http://www.precisejava.com/javaperf/j2ee/EJB.htm
EJB performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

- EJB calls are expensive. A method call from the client could cover all the following: get Home reference from the NamingService (one network round trip); get EJB reference (one or two network roundtrips plus remote creation and initialization of Home and EJB objects); call method and return value on EJB object (two or more network rountrips: client-server and [mutliple] server-db; several costly services used such as transactions, persistence, security, etc; multiple serializations and deserializations).
- If you don't need EJB services for an object, use a plain Java object and not an EJB object.
- Use Local interfaces (from EJB2.0) if you deploy both EJB Client and EJB in the same JVM. (For EJB1.1 based applications, some vendors provide pass-by-reference EJB implementations that work like Local interfaces).
- Wrap multiple entity beans in a session bean to change multiple EJB remote calls into one session bean remote call and several local calls (pattern called SessionFacade).
- Change multiple remote method calls into one remote method call with all the data combined into a parameter object.

- Control serialization by modifying unnecessary data variables with 'transient' key word to avoid unnecessary data transfer over network.
- Cache EJBHome references to avoid JNDI lookup overhead (pattern called ServiceLocator).
- Declare non-transactional methods of session beans with 'NotSupported' or 'Never' transaction attributes (in the ejb-jar.xml deployment descriptor file).
- Transactions should span the minimum time possible as transactions lock database rows.
- Set the transaction time-out (in the ejb-jar.xml deployment descriptor file).
- Use clustering for scalability.
- Tune the EJB Server thread count.
- Use the HttpSession object rather than a Stateful session bean to maintain client state.
- Use the ECperf benchmark to help differentiate EJB server performances.
- Tune the Stateless session beans pool size to minimize the creation and destruction of beans.
- Use the setSessionContext() or ejbCreate() method to cache bean specific resources. Release acquired resources in the ejbRemove() method.
- Tune the Stateful session beans cache size to and time-out minimize activations and passivations.
- Allow stateful session beans to be removed from the container cache by explicitly using the remove() method in the client.
- Tune the entity beans pool size to minimize the creation and destruction of beans.
- Tune the entity beans cache size to minimize the activation and passivation of beans (and associated database calls).
- Use the setEntityContext() method to cache bean specific resources and release them from the unSetEntityContext() method.
- Use Lazy loading to avoid unnecessary pre-loading of child data.
- Choose the lowest cost transaction isolation level that avoids corrupting the data. Transaction levels in increasing cost are: TRANSACTION_READ_UNCOMMITED, TRANSACTION_READ_COMMITED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE.
- Use the lowest cost locking available from the database that is consistent with any transaction.
- Create read-only entity beans for read only operations.
- Use a dirty flag where supported by the EJB server to avoid writing unchanged EJBs to the database.
- Commit the data after the transaction completes rather than after each method call (where supported by EJB server).
- Do bulk updates to reduce database calls.
- Use CMP rather than BMP to utilize built-in performance optimization facilities of CMP.
- Use ejbHome() methods for global operations (from EJB2.0).
- Tune the connection pool size to minimize the creation and destruction of database connections.
- Use JDBC directly rather than using entity beans when dealing with large amounts of data such as searching a large database.
- Combine business logic with the entity bean that holds the data needed for that logic to process.
- Tune the Message driven beans pool size to optimize the concurrent processing of messages.
- Use the setMesssageDrivenContext() or ejbCreate() method to cache bean specific resources, and release those resources from the ejbRemove() method.

http://www.precisejava.com/javaperf/j2ee/JDBC.htm
JDBC performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

- Use the fastest driver available to the database: normally type 4 (preferably) or type 3.
- Tune the defaultPrefetch and defaultBatchValue settings.
- Get database connections from a connection pool: use javax.sql.DataSource for optimal configurability. Use the vendor's connection pool; or ConnectionPoolDataSource and PooledConnection from JDBC2.0; or a proprietary connection pool.
- Batch your transactions. Turn off autocommit and explicitly commit a set of statements.
- Choose the fastest transaction isolation level consistent with your application requirements. Levels from fastest to slowest are: TRANSACTION_NONE, TRANSACTION_READ_UNCOMMITED, TRANSACTION_READ_COMMITED, TRANSACTION_REPEATABLE_READ, TRANSACTION_SERIALIZABLE.
- Close resources (e.g. connections) when finished with them.
- Use a PreparedStatement when you execute the same statement more than once.
- Use CallableStatement to execute stored procedures. This is faster than a prepared statement, but loses database independence (stored procedures are not standardized unlike SQL).
- Batch updates and accesses with Statements and ResultSets (with executeBatch() and setFetchSize()).
- Set up the proper direction for processing rows.
- Use the proper getXXX() methods.
- Write SQL queries that minimize the data returned.
- Cache read-only and read-mostly tables data.
- Use the Page-by-Page Iterator pattern to repeatedly pass small amounts of data rather than huge chunks.

http://www.precisejava.com/javaperf/j2ee/Servlets.htm
Servlet performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

- Use the servlet init() method to cache static data, and release them in the destroy() method.
- Use StringBuffer rather than using + operator when you concatenate multiple strings.
- Use the print() method rather than the println() method.
- Use a ServletOutputStream rather than a PrintWriter to send binary data.
- Initialize the PrintWriter with the optimal size for pages you write.
- Flush the data in sections so that the user can see partial pages more quickly.
- Minimize the synchronized block in the service method.
- Implement the getLastModified() method to use the browser cache and the server cache.
- Use the application server's caching facility.
- Session mechanisms from fastest to slowest are: HttpSession, Hidden fields, Cookies, URL rewriting, the persistency mechanism.
- Remove HttpSession objects explicitly in your program whenever you finish the session.
- Set the session time-out value as low as possible.
- Use transient variables to reduce serialization overheads.
- Disable the servlet auto reloading feature.
- Tune the thread pool size.

http://www.onjava.com/pub/a/onjava/2002/07/17/web.html
High load web servlets (Page last updated July 2002, Added 2002-07-24, Author Pier Fumagalli, Publisher OnJava). Tips:

- Hand off requests for static resources directly to the web server by specifying the URL, not by redirecting from the servlet.
- Use separate webservers to deliver static and dynamic content.

- Cache as much as possible. Make sure you know exactly how much RAM you can spare for caches, and have the right tools for measuring memory.
- Load balance the Java application using multiple JVMs.
- Use ulimit to monitor the number of file descriptors available to the processes. Make sure this is high enough.
- Logging is more important than the performance saved by not logging.
- Monitor resources and prepare for spikes.

http://www.precisejava.com/javaperf/j2ee/JSP.htm
JSP performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

- Use the jspInit() method to cache static data, and release them in the jspDestroy() method.
- Use the jspInit() method to cache static data.
- Use StringBuffer rather than using + operator when you concatenate multiple strings.
- Use the print() method rather than the println() method.
- Use a ServletOutputStream rather than a PrintWriter to send binary data.
- Initialize the PrintWriter with the optimal size for pages you write.
- Flush the data in sections so that the user can see partial pages more quickly.
- Minimize the synchronized block in the service method.
- Avoid creating a session object with the directive <%@ page session="false" %>
- Increase the buffer size of System.out with the directive <%@ page buffer="12kb" %>
- Use the include *directive* instead of the include *action* when you want to include another page.
- Minimize the scope of the 'useBean' action.
- Custom tags incur a performance overhead. Use as few as possible.
- Use the application server's caching facility, and the session and application objects (using getAttribute()/setAttribute()). There are also third-party caching tags available.
- Session mechanisms from fastest to slowest are: session, Hidden fields, Cookies, URL rewriting, the persistency mechanism.
- Remove 'session' objects explicitly in your program whenever you finish the session.
- Reduce the session time-out as low as possible.
- Use 'transient' variables to reduce serialization overheads.
- Disable the JSP auto reloading feature.
- Tune the thread pool size.

http://www.precisejava.com/javaperf/j2ee/JMS.htm
JMS performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

- Start the consumer before you start the producer so that the initial messages do not need to queue.
- Use a ConnectionConsumer to process messages concurrently with a ServerSessionPool.
- Close resources (e.g. connections, session objects, producers, consumers) when finished with them.
- DUPS_OK_ACKNOWLEDGE and AUTO_ACKNOWLEDGE perform better than CLIENT_ACKNOWLEDGE.
- Use separate transactional sessions and non-transactional sessions for transactional and non-transactional messages.
- Tune the Destination parameters: a smaller capacity increases message throughput; a higher redelivery delay and lower redelivery limit reduces the overhead.

- Choose non-durable (NON_PERSISTENT) messages wherever appropriate to avoid the persistency overhead.
- Set the TimeToLive value as low as feasible (default is for messages to never expire).
- Receive messages asynchronously with a MessageListener implementation.
- Choose the message type that minimizes memory overheads.
- Use 'transient' variables to reduce serialization overheads.

http://www.precisejava.com/javaperf/j2ee/Patterns.htm
Pattern performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

- The ServiceLocator/EJBHomeFactory Pattern reduces the expensive JNDI lookup process by caching EJBHome objects.
- The SessionFacade Pattern reduces network calls by combining accesses to multiple Entity beans into one access to the facade object.
- The MessageFacade/ServiceActivator Pattern moves method calls into a separate object which can execute asynchronously.
- The ValueObject Pattern combines remote data into one serializable object, thus reducing the number of network transfers required to access multiple items of remote data.
- The ValueObjectFactory/ValueObjectAssembler Pattern combines remote data from multiple remote objects into one serializable object, thus reducing the number of network transfers required to access multiple items of remote data.
- The ValueListHandler Pattern: avoids using multiple Entity beans to access the database, using Data Access Objects which explicitly query the database; and returns the data to the client in batches (which can be terminated) rather than in one big chunk, according to the Page-by-Page Iterator pattern.
- The CompositeEntity Pattern reduces the number of actual entity beans by wrapping multiple java objects (which could otherwise be Entity beans) into one Entity bean.

http://softwaredev.earthweb.com/java/article/0,,12082_862481,00.html
Writing a seamless audio looper (Page last updated August 2001, Added 2001-08-20, Author Greg Travis, Publisher EarthWeb). Tips:

- Switching audio streams from one piece of sound to another requires some fiddly managing of the transition delay in order to avoid a gap in the audio output.
- To avoid the transition delay, you need to: flush the output buffer; find out how much data was dumped; add a fudge factor; and combine these values to determine from where to start playing the new audio stream.

http://www.sys-con.com/java/article.cfm?id=1307
Generating code dynamically (Page last updated February 2002, Added 2002-02-22, Author Norman Richards, Publisher Java Developers Journal). Tips:

- Compiling code into classes at runtime, such as for JSP pages, provides excellent flexibility with almost no performance overhead.
- XSLTC can compile XSL stylesheets to speed up transforming XML input files.
- If a complex interpreted procedure is expected to be used more than once, it can be more efficient to convert the procedure into an expression tree which will apply the procedure optimally.
- Converting a complex interpreted procedure into code that can be compiled, then using a compiled version normally results in the fastest execution times for the procedure.
- Sun's javac is not a very efficient compiler. Faster compilers are available, such as jikes.

- Compiling code at runtime can take a significant amount of time. If the compile time needs to be minimized, it is important to use the fastest compiler available.
- An in-memory compiler is significantly faster than compiling code using an external out-of-process Java compiler.
- Generating bytecode directly in-process is significantly faster than compiling code using an external out-of-process Java compiler, and is also faster than using an in-memory compiler. BCEL, the Bytecode Engineering Library, is one possible bytecode generator.

http://www.j3d.org/tutorials/quick_fix/perf_guide_1_1.html
Java 3D performance tips (Page last updated June 2001, Added 2001-08-20, Author Doug Twilleager, Publisher J3D). Tips:

- Once an application calls BranchGroup.compile() or SharedGroup.compile(), only objects with their capability bits set can be modified.
- Use capability bits to describe which objects change at runtime, so that J3D can optimize the app. Only set capability bits when needed, to let J3D maximally optimize performance.
- Set the bounds of objects so that J3D can ignore objects outside target object spatial scopes.
- Reorder leaf nodes for the most efficient rendering.
- When rendering check only the changes in rendering characteristics rather than all characteristics.
- Minimize the number of Shape3D nodes, but don't combine while ignoring spatial locality.
- Use the stripifier, or manually stripify the application: try to convert the geometry into long strips of triangles rather than fans of triangles.
- Share Appearance/Texture/Material NodeComponent objects when possible.
- Set the thread priorities appropriately, or use the default priority. Minimize thread activity.
- Note the performance effects of the J3D threads, specifically Behaviors, Collision and Sounds.
- J3D fully supports multi-processor machines. Use native threads where possible.
- Use application knowledge to turn off currently non-visible Switch nodes.
- Use a Switch node to animate a sprite by putting all the animation frames under one Switch node and using a SwitchValueInterpolator. This increases memory consumption in favor of smooth animations.
- Unordered groups are faster than ordered groups.
- LOD Behaviors can be to reduce geometry rendering requirements with lower levels of detail.
- Use bounds based picking rather than geometry based picking.
- Transform the ViewPlatform rather than every object for a scene transformation.

http://www.javaworld.com/javaworld/jw-07-2002/jw-0703-service.html
The Verified Service Locator pattern (Page last updated July 2002, Added 2002-07-24, Author Paulo Caroli, Publication JavaWorld, Publisher JavaWorld). Tips:

- The Service Locator pattern improves performance by caching service objects that have a high-lookup cost.
- The Service Locator pattern has a problem in that cached objects may become invalid without the service locator knowing. The Verified Service Locator pattern periodically tests the validity of the caches objects to avoid providing invalid service objects to requestors.

http://developer.java.sun.com/developer/community/chat/JavaLive/2002/jl0515.html
Sun Community chat on Java BluePrints (Page last updated May 2002, Added 2002-07-24, Author Edward Ort, Publication Sun Developer, Publisher Sun). Tips:

- For very large transactions, use transaction attribute TX_REQUIRED for EJB methods to have all the method calls in a call chain use the same transaction.
- Make tightly coupled components local to each other. Put remote beans primarily as facades across subsystems.
- The page-by-page pattern is designed to handle cases where the result set is large, and the end-user is not interested in seeing all of the results. There is really no upper threshold for the size of result set in the pattern.

http://www.onjava.com/pub/a/onjava/2002/07/10/jboss.html
Clustering with JBoss (Page last updated July 2002, Added 2002-07-24, Authors Bill Burke, Sacha Labourey, Publisher OnJava). Tips:

- A hardware- or software-based HTTP load-balancer usually sits in front of the application servers within a cluster. The load balancer can decrypt HTTPS requests and distribute load.
- HTTP session replication is expensive for a J2EE application server. If you can live with forcing a user to log in again after a server failure, then an HTTP load-balancer probably provides all of the fail-over and load-balancing functionality you need.
- If you are storing things other than EJB Home references in your JNDI tree, then you may need clustered JNDI.
- 24/7 availability needs the ability to hot-deploy and undeploy new applications and new versions, and to apply patches, without bringing down the application server for maintenance.
- Smart proxies can be used to implement load-balancing and fail-over for EJB remote clients. These proxies manage a list of available RMI connections one of which it will use to service an invocation.

http://java.ittoolbox.com/pub/SC071902/httprevealer_servlets_itx.htm
Speeding web page downloads using compression (Page last updated July 2002, Added 2002-07-24, Author Steven Chau, Publication HttpRevealer.com, Publisher HttpRevealer.com). Tips:

- Browsers sending "Accept-Encoding: gzip" will accept gziped compressed pages. Return the page compressed with "Content-Encoding: gzip" using GZIPOutputStream.
- Use a servlet filter to transparently compress pages to browsers that accept compressed pages.

http://www.theserverside.com/resources/article.jsp?l=Prepared-Statments
Optimizing JDBC Prepared Statments. Also a followup discussion at http://www.theserverside.com/discussion/thread.jsp?thread_id=8013 (Page last updated July 2001, Added 2001-08-20, Author ?, Publisher The Server Side). Tips:

- Databases analyze query statements to decide how to process them most optimally, then cache the resulting query plan, keyed on the full statement. Reusing identical statements reuses the query plan.
- Altering the statement causes a new query plan to be generated for each new statement. However statements with parameters can have the query plan reused, so use parameters rather than regenerating the statement with different values.
- Using a new connection requires a prepared statement to be recreated. Reusing connections allows a prepared statement to be reused.
- Connection pools should have associated PreparedStatement caches so that the PreparedStatements are automatically reused.

http://billharlan.com/pub/papers/Improving_Swing_Performance.html
Swing performance tips (Page last updated 1999, Added 2001-05-21, Author Bill Harlan, Publisher Harlan). Tips:

- Redraw events can easily be generated faster than the redraw can execute. Ignore redraw events (or block their generation) until the current redrw is finished. Don't up redraw events.
- Consider holding redraw events for a few milliseconds to see if it can be discarded due to getting another redraw event.
- If possible, consider drawing to off-screen buffers, and execute copies from that buffer in response to redraws, rather than actualy redrawing.
- Extend from JPanel, not Canvas; override paintComponent(), not paint().
- Action listeners are all executed in the *one* event-dispatching thread. Time-consuming listeners should execute their work in a separate thread and should avoid blocking the event-dispatching thread. (To reenter the event-dispatching thread calling SwingUtilities.invokeLater() or invokeAndWait()).
- Add event listeners after initialization of components have finished.

http://java.sun.com/products/jfc/tsc/articles/performance/index.html
Swing performance tips (Page last updated March 2001, Added 2001-05-21, Author Steve Wilson, Publisher Sun). Tips:

- Use the latest version of Swing available, as the Swing development team have an ongoing project tp improve performance.
- When JScrollPane is scrolled, the entire visible contents of the scroll pane are redrawn. A backing store (off screen buffer) can be enabled using setBackingStoreEnabled(true) to speed up redraws, but this has some limitations: an extra buffer to copy can be significant for simple drawing operations; the backing store doesn't work when scrollRectToVisible() is called directly by the programmer (depends on Swing version); extra RAM is needed to maintain the extra backing buffer.
- Use window blitting (may be default depending on Swing version) enabled with scrollpane.getViewport().putClientProperty("EnableWindowBlit", Boolean.TRUE).
- Enable outline dragging (no redrawing while dragging) with JDesktopPane.putClientProperty("JDesktopPane.dragMode", "outline").
- Enable faster dragging using blitting with JDesktopPane.putClientProperty("JDesktopPane.dragMode", "faster").

http://www.onjava.com/pub/a/onjava/excerpt/JavaRMI_10/index.html
Chapter 10, "Serialization" from "Java RMI" (Page last updated November 2001, Added 2001-12-26, Author William Grosso, Publisher OnJava). Tips:

- Use transient to avoid sending data that doesn't need to be serialized.
- Serialization is a generic marshalling mechanism, and generic mechanisms tend to be slow.
- Serialization uses reflection extensively, and this also makes it slow.
- Serialization tends to generate many bytes even for small amounts of data.
- The Externalizable interface is provided to solve Serialization's performance problems.
- Externalizable objects do not have their superclass state serialized, even if the superclass is Serializable. This can be used to reduce the data written out during serialization.
- Use Serializable by default, then make classes Externalizable on a case-by-case basis to improve performance.

http://www.stqemagazine.com/featured.asp?id=10
Web application scalability. (Page last updated June 2000, Added 2001-05-21, Author Billie Shea, Publisher STQE Magazine). Tips:

- Web application scalability is the ability to sustain the required number of simultaneous users and/or transactions, while maintaining adequate response times to end users.
- The first solution built with new skills and new technologies will *always* have room for improvement.
- Avoid deploying an application server that will cause embarrassment, or that could weaken customer confidence and business reputation [because of bad response times or lack of calability].
- Consider application performance throughout each phase of development and into production.
- Performance testing must be an integral part of designing, building, and maintaining Web applications.
- There appears to be a strong correlation between the use of performance testing tools and the likelihood that a site would scale as required.
- Automated performance tests must be planned for and iteratively implemented to identify and remove bottlenecks.
- Validate the architecture: decide on the maximum scaling requirements and then performance test to validate the necessary performance is achievable. This testing should be done on the prototype, before the application is built.
- Have a clear understanding of how easily your configurations of Web, application, and/or database servers can be expanded.
- Factor in load-balancing software and/or hardware in order to efficiently route requests to the least busy resource.
- Consider the effects security will have on performance: adding a security layer to transactions will impact response times. Dedicate specific server(s) to handle secure transactions.
- Select performance benchmarks and use them to quantify the scalability and determine performance targets and future performance improvements or degradations. Include all user types such as "information-gathering" visitors or "transaction" visitors in your benchmarks.
- Perform "Performance Regression Testing": continuously re-test and measure against the established benchmark tests to ensure that application performance hasn?t been degraded because of the changes you?ve made.
- Performance testing must continue even after the application is deployed. For applications expected to perform 24/7 inconsequential issues like database logging can degrade performance. Continuous monitoring is key to spotting even the slightest abnormality: set performance capacity thresholds and monitor them.
- When application transaction volumes reach 40% of maximum expected volumes, it is time to start executing plans to expand the system

http://www.stqemagazine.com/featured.asp?id=15
Web Load Test Planning (Page last updated April 2001, Added 2001-05-21, Author Alberto Savoia, Publisher STQE Magazine). Tips:

- The only reliable way to determine a system?s scalability is to perform a load test in which the volume and characteristics of the anticipated traffic are simulated as realistically as possible.
- It is hard to design and develop load tests that come close to matching real loads.
- Characterize the anticipated load as objectively and systematically as possible: use existing log files where possible; characterize user sessions (pages viewed - number and types;

duration of session; etc). Determine the range and distribution of variations in sessions. Don't use averages, use representative profiles.
- Estimate target load and peak levels: estimate overall and peak loads for the server and expected growth rates.
- Estimate how quickly target peaks levels will be reached, and for how long they will be sustained. The duration of the peak is important and the server must be designed to handle it.
- The key elements of a load test design are: test objective (e.g. can the server handle N sessions/hr peak load level?); pass/fail criteria (e.g. pass if response times stay within define values); script description (e.g. user1: page1, page2, ...; user2: page1, page3, start transaction1, etc); scenario description (which scripts at which frequency, and how load increases).

http://www.theserverside.com/resources/articles/JSP-Performance/ProJsp.html
Performance chapter (chapter 20) from "Professional JSP 2nd Edition" (Page last updated August 2001, Added 2001-10-22, Author Simon Brown, Robert Burdick, Darko Cokor, Jayson Falkner, Ben Galbraith, RodJohnson, Larry Kim, Casey Kochmer, Thor Kristmundsson, Sing Li, Dan Malks, Mark Nelson, Grant Palmer, Bob Sullivan, Geoff Taylor, John Timney, Sameer Tyagi, Geert Van Damme, Steve Wilkinson, Publisher The Server Side). Tips:

- The user's view of the response time for a page view in his browser depends on download speed and on the complexity of the page. e.g. the number of graphics. A poorly-designed highly graphical dynamic website could be seen as 'slow' even if the web downloads are individually quite fast.
- No web application can handle an unlimited number of requests; the trick in optimization is to anticipate the likely user demand and ensure that the web site can gracefully scale up to the demand while maintaining acceptable levels of speed.
- Profile the server to identify the bottlenecks. Note that profiling can be done by instrumenting the code with measurement calls if a profiler is unavailable.
- One stress test methodology is: determine the maximum acceptable response time for getting a page; estimate the maximum number of simultaneous users; simulate user requests, gradually adding simulated users until the web application response delay becomes greater than the acceptable response time; optimize until you reach the desired number of users.
- Pay special attention to refused connections during your stress test: these indicate the servlet is overwhelmed.
- There is little performance penalty to using an MVC architecture.
- Use resource pools for expensive resources (like database connections).
- Static pages are much faster than dynamic pages, where the web server handles static pages separately.
- Servlet filtering has a performance cost. Test to see if it is an acceptable cost.
- Ensure that the webserver is configured to handle the expected number of user for example: enough ready sockets; enough disk space; enough CPU.
- Use the fastest JVM you have access to.

http://developer.java.sun.com/developer/Books/performance2/chap3.pdf
Chapter 3 of "High Performance Java Computing : Multi-Threaded and Networked Programming", "Race Conditions and Mutual Exclusion" (Page last updated January 2001, Added 2001-02-21, Authors George Thiruvathukal, Thomas Christopher, Publisher Sun). Tips:

- Execute I/O in blocks rather than one byte at a time.
- I/O reads are normally faster than writes. This means that I/O performance can be improved by decoupling reading and writing to dedicated threads, rather than interleaving reads and writes.

- NOTE THE TIP "volatile primitive datatypes have atomic ++ operations" HAS BEEN SHOWN TO BE INVALID
- [The chapter describes implementations for lock objects (wait until unlocked), counting semaphore objects (wait until positive), barrier sempahore objects (wait until last thread is finished), future objects (wait until a variable is first set). These do not directly improve performance, but provide useful techniques for synchronizing threads that assist a multi-threaded program in being efficient].
- Use resource enumeration (acquire resources in a set order) to avoid deadlocks.

http://developer.java.sun.com/developer/Books/performance2/chap4.pdf
Chapter 4 of "High Performance Java Computing : Multi-Threaded and Networked Programming", "Monitors" (Page last updated January 2001, Added 2001-02-21, Authors George Thiruvathukal, Thomas Christopher, Publisher Sun). Tips:

- Java monitors are not necessarily the most efficient synchronization mechanism, especially if transferring the lock can lead to a race condition [chapter discusses a more complete Monitor class].
- `volatile` fields can be slower than non-`volatile` fields, because the system is forced to store to memory rather than use registers. But they may useful to avoid concurrency problems.
- [The chapter discusses various policies for synchronizing threads trying to read from or write to shared resources, which provide different scheduling policies: one thread at a time; readers-preferred (readers have priority); writers-preferred (writers have priority); alternating readers-writers (alternates between a single writer and a batch of readers); take-a-number (first-come, first-served)].

http://www.sys-con.com/java/article.cfm?id=639
Benchmarking JMS (Page last updated March 2001, Added 2001-03-21, Author Dave Chappell, Bill Wood, Publisher Java Developers Journal). Tips:

- Scaling middleware exposes a number of issues such as threading contention, network bottlenecks, message persistence issues, memory leaks, and overuse of object allocations.
- [Article dicusses questions to ask when setting up benchmarks for messaging middleware].
- Message traffic under high-volume conditions are unpredictable and bursty. Messages can be produced far faster than they can be consumed, causing congestion. This condition requires the message sends to be throttled with flow control (could be an exception, or an automatic resend).
- When testing performance, run overnight and over weekends to generate longer term trends. Some concerns are: testing without a real network connection can give false measures; low user simulation can be markedly different from high user simulations; network throughput may be large than the deployed environment; nonpersistent message performance is dependent on processor and memory; disk speed is crucial for persistent messages.
- [Article provides a benchmark harness for testing JMS].

http://www.javaworld.com/javaworld/jw-02-2001/jw-0223-performance.html
Designing Java Performance: reducing object creation (Page last updated March 2001, Added 2001-03-21, Author Brian Goetz, Publisher JavaWorld). Tips:

- Watch out for method interfaces which force unnecessary or inefficient object creation.
- Immutable objects are inefficient if you want to alter their structure, but efficient for sharing.

- One way to avoid creating objects simply for information is to provide finer-grained methods which return information as primitives. This swaps object creation for increased method calls.
- A second technique to avoid creating objects is to provide methods which accept dummy information objects that have their state overwritten to pass the information.
- A third technique to avoid creating objects is to provide immutable classes with mutable subclasses, by having state defined as `protected` in the superclass, but with no public updators. The subclass provides public updators, hence making it mutable.
- Don't try to speed up the application if there is no performance problem.

http://theserverside.com/home/thread.jsp?thread_id=3276
Some performance tips (Page last updated January 2001, Added 2001-01-19, Author Shyam Lingegowda, Publisher The Server Side). Tips:

- Use buffering for files & stream i/o . Use byte streams (not char-streams) for ASCII characters.
- Recycle objects wherever possible.
- Factor out constant computations from loops. For Servlets, push one time computations into the init() method.
- Use StringBuffer when dealing with mutable strings. Initialize the StringBuffer with proper size.
- Let the compiler do compile time string concatenation.
- Comparison of two string objects is faster if they differ in length.
- StringTokenizer is slow.
- minimize the number of objects you create.
- Avoid initializing twice.
- Order boolean expressions so that they execute as fast as possible.
- ArrayList is faster than Vector.
- Minimize calls to Date and related classes.

http://www.onjava.com/pub/a/onjava/2002/01/09/dataexp1.html
Expiring cached data (Page last updated January 2001, Added 2002-01-25, Author William Grosso, Publisher OnJava). Tips:

- Caching data on the client can improve performance, reduce communication overheads and increase the scalability of an application.
- Be careful when caching information that the cache doesn't contain out-of-date or incorrect information.
- Servlet sessions expire after a settable timeout, but screens that automatically refresh can keep a session alive indefinitely, even when the screen is no longer in use.
- Database connection pools can take one of two strategies: a limited size pool, where attempts to make connections beyond the pool size must wait for a connection to become idle; or a flexible sized pool with a preferred size which removes idle connections as soon as the preferred size is exceeded (i.e. temporarily able to exceed the preferred size). The fixed size pool is generally considered to be the better choice.
- A time-based expiration strategy is appropriate for most types of cache elements. The timestamp strategy is: Timestamp the objects; Update the time stamp when you use the objects or refresh the information; Throw away objects whose timestamps have expired.
- Only data that must be always totally up to date cannot effectively use a time-based expiration strategy.
- [Article discusses and implements a time-based expiration framework].

http://portals.devx.com/Nokia/Article/6218
J2ME game building (Page last updated April 2002, Added 2002-05-19, Author Dale Crowley, Publisher DevX). Tips:

- J2ME device memory and speeds are very limited which affects everything from the data-loading speed to the frame/refresh rate, and seriously limits the ability to animate characters or otherwise rapidly change the screen.
- Smart graphics is important: you need to draw clear, concise images at extremely low resolutions and with very small palettes. Animated characters need dynamic, easily-read poses which avoid kicks looking like a dance steps, or punches looking like an arm waves.
- Use public variables in your classes, rather than using accessors. This is technically bad programming practice but it saves bytecode space.
- Be extra careful to place things in memory only when they are in use. For example, discard an introduction splash screen after display.
- Try to reduce the number of classes used. Combine classes into one if they vary only slightly in behavior. Every class adds size overheads.
- Remember that loading and installing applications into J2ME phones is a relatively slow process.

http://developer.java.sun.com/developer/community/chat/JavaLive/2002/jl0423.html
Sun community chat on High Performance GUIs with the JFC/Swing, with Steve Wilson, Scott Violet, and Chet Haase (Page last updated April 2002, Added 2002-05-19, Author Edward Ort, Publisher Sun). Tips:

- [Some discussion of performance improvents in 1.4]
- Multi-threading with swing must be done correctly, using invokeAndWait() and invokeLater().
- Default models have performance limitations. Create dedicated models for high performance. Consider using a custom RepaintManager for very large tables.
- Don't use a MouseListener with a renderer
- BufferedImage is treated more optimally than MemoryImageSource
- Try using createImage(w,h), which returns an image in the same format as the screen, which allows faster copies from that image to the screen (important for copying speed issues).

http://developer.java.sun.com/developer/JDCTechTips/2002/tt0709.html
LinkedHashMap and RandomAccess (Page last updated July 2002, Added 2002-07-24, Author Glen McCluskey, Publisher Sun). Tips:

- LinkedHashMap preserves various ordering information, optionally including access ordering which makes LinkedHashMap appropriate for a least recently used (LRU) cache.
- ArrayList has fast random access of elements, LinkedList has slow random access of elements. List classes that implement the RandomAccess interface have fast random access and using get() to iterate their elements is efficient. If RandomAccess is not implemented, use an Iterator to iterate the elements.

http://www.onjava.com/pub/a/onjava/2002/01/30/dataexp2.html
Data expiration in caches (Page last updated January 2002, Added 2002-02-22, Author William Grosso, Publisher OnJava). Tips:

- [Article discusses and implements a framework for a cache with built in element expiration handling].

http://www.sys-con.com/java/article.cfm?id=1534
Emulating another system (a ZX Spectrum) (Page last updated July 2002, Added 2002-07-24, Author Razvan Surdulescu, Publisher Java Developers Journal). Tips:

- Painting pixel by pixel by repeatedly calling fillRect() is slow. Instead create the offscreen image as a decorator for a java.awt.image.MemoryImageSource object containing a byte array in RGB format with the pixel data. The rendering code updates the byte array and then calls MemoryImage-Source.newPixels() to notify the object that the data has been updated.
- Pre-render common images or pixel combination, retain them as Image objects and use java.awt.Graphics.drawImage() (Java 1) or java.awt.image.BufferedImage.setRGB() (Java 2) to render the image to the graphics buffer.

http://www.aceshardware.com/read.jsp?id=45000251
Report of how Ace's Hardware made their SPECmine tool blazingly fast (Page last updated December 2001, Added 2002-02-22, Author Chris Rijk, Publisher Ace's Hardware). Tips:

- Tranform your data to minimize the costs of searching it.
- If your dataset is small enough, read it all into memory or use an in-memory database (keeping the primary copy on disk for recovery).
- An in-memory datavase avoids the following overheads: no need to pass data in from a separate process; less memory allocation by avoiding all the data copies as it's passed between processes and layers; no need for data conversion; fine-tuned sorting and filtering possible; other optimizations become simpler.
- Pre-calculation makes some results faster by making the database data more efficient to access (by ordering it in advance for example), or by setting up extra data in advance, generated from the main data, to make calculating the results for a query simpler.
- Pre-determine possible data values in queries, and use boolean arrays to access the chosen values.
- Pre-calculate all formatting that is invariant for generated HTML pages. Cache all reused HTML fragments.
- Caching many strings may consume too much memory. IF memory is limited, it may be more effective to generate strings as needed.
- Write out strings individually, rather than concatenating them and writing the result.
- Extract common strings into an identical string object.
- Compress generated html pages to send to the user, if their browser supports compressed html. This is a heavier load on the server, but produces a significantly faster transfer for limited bandwidth clients.
- Some pages are temporarily static. Cache these pages, and only re-generate them when they change.
- Caching can significantly improve the responsiveness of a website.

http://lists.xml.org/archives/xml-dev/200201/msg00477.html
Email summarizing best practices for Promoting Scalable Web Services (Page last updated January 2002, Added 2002-02-22, Author Roger L. Costello, Publisher Costello). Tips:

- Web services best practices are mainly the same as guidelines for developing other distributed systems.
- Stay away from using XML messaging to do fine-grained RPC, e.g. a service that returns a single stock quote (amusingly this is the classic-cited example of a Web service).
- Do use course-grained RPC, that is, use Web services that "do a lot of work, and return a lot of information".

- When the transport may be slow and/or unreliable, or the processing is complex and/or long-running, consider an asynchronous messaging model.
- Always take the overall system performance into account. Don't optimize until you know where the bottlenecks are, i.e., don't assume that XML's "bloat" or HTTP's limitations are a problem until they are demonstrated in your application.
- Take the frequency of the messaging into account. Replicate data as necessary.
- For aggregation services, try to retrieve data during off-hours in large, course-grained transactions.

http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-soap.html
Caching SOAP services (Page last updated March 2002, Added 2002-03-25, Author Ozakil Azim and Araf Karsh Hamid, Publisher JavaWorld). Tips:

- Repeated SOAP-client calls to access server state can choke a network and degrade the server performance. Cache data on the client whenever possible to avoid requests to the server.
- Ensure the client data remains up to date by using a call to a server service which blocks until data is changed.

http://developer.java.sun.com/developer/JDCTechTips/2002/tt0305.html
String concatenation, and IO performance. (Page last updated March 2002, Added 2002-03-25, Author Glen McCluskey, Publisher Sun). Tips:

- String concatenation '+' is implemented by the Sun compiler using StringBuffer, but each concatenation creates a new StringBuffer so is inefficient for multiple concatenations.
- Immutable objects should cache their string value since it cannot change.
- Operating systems can keep files in their own file cache in memory, and accessing such a memory-cached file is much faster than accessing from disk. Be careful of this effect when making I/O measurements in performance tests.
- Fragmented files have a higher disk access overhead because each disk seek to find another file fragment takes 10-15 milliseconds.
- Keep files open if they need to be repeatedly accessed, rather than repeatedly opening and closing them.
- Use buffering when accessing file contents.
- Explicit buffering (reading data into an array) gives you direct access to the array of data which lets you iterate over the elements more quickly than using a buffered wrapper class.
- Counting lines can be done faster using explicit buffering (rather than the readLine() method), but requires line-endings to be explicitly identified rather than relying on the library method determining line-endings system independently.

http://developer.java.sun.com/developer/community/chat/JavaLive/2002/jl0305.html
Sun community chat on EJBs with Pravin Tulachan (Page last updated March 2002, Added 2002-03-25, Author Edward Ort, Publisher Sun). Tips:

- CMP (container managed persistence) is generally faster than BMP (bean managed persistence).
- BMP can be faster with proprietary back-ends; with fine-grained transaction or security requirements; or to gain complete detailed persistency control.
- Scalability is improved by passing primary keys rather than passing the entities across the network.
- EJB 2.0 CMP is far faster than EJB 1.1 CMP. EJB 1.1 CMP was not necessarily capable of scaling to high transaction volumes.

- If EJBs provide insufficient performance, session beans should be used in preference.
- Don't make fine-grained method calls across the network. Use value object and session facade design patterns instead.

http://softwaredev.earthweb.com/java/article/0,,12082_951051,00.html
Multithreading and read-write locks (Page last updated January 2002, Added 2002-01-25, Author Nasir Khan, Publisher EarthWeb). Tips:

- When a thread passes through a synchronized block, *all* variables throughout the thread are synchronized with main memory, not just the set of variables in the current method.

http://softwaredev.earthweb.com/java/article/0,,12082_951051,00.html
Multithreading and read-write locks, part 2 (Page last updated January 2002, Added 2002-01-25, Author Nasir Khan, Publisher EarthWeb). Tips:

- Operations on primitive variables are atomic (except double and long), but a combination of two atomic operations is not atomic, and it is easy to make a mistake about this.
- Volatile variables are always synchronized with the main memory copy.

http://www.javaworld.com/javaworld/jw-01-2002/jw-0111-hotspotgc.html
Hotspot garbage collection in detail (Page last updated January 2002, Added 2002-01-25, Author Ken Gottry, Publisher JavaWorld). Tips:

- HotSpot garbage collection default parameters are effective for most small applications, but can be tuned to improve throughput for large, server-side applications.
- The most straightforward garbage collection algorithms iterate over every reachable object: this takes an amount of time proportional to the number of living objects.
- Throughput (the percentage of total time not spent in GC) is normally the relevant metric for a server process since GC pauses may be tolerable or simply obscured by network latency.
- Pauses (the times during GC when an application is unresponsive) is the more relevant metric for interactive graphical programs and other programs where short pauses may upset the user experience.
- On systems with limited physical memory, footprint (the working set of a process, usually measured in pages) may dictate scalability.
- [Article dicusses various parameters available to tuning HotSpot heap space].
- Use verbosegc to capture garbage collection statistics.

http://www-106.ibm.com/developerworks/webservices/library/ws-quality.html
Quality of service for web services (Page last updated January 2002, Added 2002-02-22, Author Anbazhagan Mani, Arun Nagarajan, Publisher IBM). Tips:

- Quality of service requirements for web services are: availability (is it running); accessiblity (can I run it now); integrity/reliability (will it crash while I run/how often); throughput (how many simultaneous requests can I run); latency (response time); regulatory (conformance to standards); security (confidentiality, authentication).
- HTTP is a best-effort delivery service. This means any request could simply be dropped. Web services have to handle this and retry.
- Web service latencies are measured in the tens to thousands of milliseconds.
- Asynchronous messaging can improve throughput, at the cost of latency.
- SOAP overheads include: extracting the SOAP envelope; parsing the contained XML information; XML data cannot be optimized very much; SOAP requires typing information

in every SOAP message; binary data gets expanded (by an average of 5-fold) when included in XML, and also requires encoding/decoding.
- Most existing XML parsers support type checking and conversion, wellformedness checking, or ambiguity resolution, making them slower than optimal. Consider using of stripped down XML parser which only pe4rforms essential parsing.
- DOM based parsers are slower than SAX based ones.
- Compress the XML when the CPU overhead required for compression is less than the network latency.
- Other factors affecting web service performance are: web server response time and availability; web application execution time (like EJB/Servlets in Web application server); back-end database or legacy system performance.
- Requests results should be cached where possible.
- Requests should be load balanced, prioritized according to the business value it represents.
- Carry out capacity planning to enable the performance to be maintained in the future.
- Extreme care should be taken to make sure that resources are not locked for long periods of time, to avoid serious scalability problems.
- Measure the performance of your web services by adding code measuring elapsed time to the generated service proxy (and recompiling). [Article gives an example].

http://www.ociweb.com/jnb/jnbMar2002.html
Object Resource Pooling (Page last updated March 2002, Added 2002-03-25, Author Paul King, Publisher OCI). Tips:

- If the overhead associated with creating a sharable resource is expensive, that resource is a good candidate for pooling.
- Pooled objects create a resource in advance and store it away so it can be reused over-and-over.
- Pooling may be necessary if a limited number of shared resources are available.
- Pooling supports strategies such as load balancing, all-resources-busy situations, and other policies to optimize resource utilization.
- [Article discusses pooling characteristics].
- Load balancing is possible by varying how pooled objects are handed out.
- Pool size can be tuned using low-water and high-water marks.
- Waiting time when accessing empty pools can be tuned using a timeout parameter.
- Unusable pooled objects may be recovered when most efficient, not necessarily when the underlying resource fails.
- The Recycler pattern fixes only the broken parts of a failed object, to minimize the replacement cost.

http://www-106.ibm.com/developerworks/java/library/j-javaio/
Using NIO (Page last updated March 2002, Added 2002-03-25, Author Aruna Kalagnanam and Balu G., Publisher IBM). Tips:

- A server that caters to hundreds of clients simultaneously must be able to use I/O services concurrently. Prior to 1.4, an almost one-to-one ratio of threads to clients made servers written in Java susceptible to enormous thread overhead, resulting in both performance problems and lack of scalability.
- The Reactor design pattern demultiplexes events and dispatches them to registered object handlers. (The Observer pattern is similar, but handles only a single source of events where the Reactor pattern handles multiple event sources).
- [Articles covers the changes needed to use java.nio to make a server efficiently muliplex non-blocking I/O from SDK 1.4].

http://www.hpmiddleware.com/newsletters/webservicesnews/features/
J2EE best practices. (Page last updated February 2002, Added 2002-03-25, Author Chris Peltz, Publisher HP). Tips:

- Executing a search against the database calls one of the finder() methods. finder() methods must return a collection of remote interfaces, not ValueObjects. Consequently the client would need to make a separate remote call for each remote interface received, to acquire data. The SessionFacade pattern suggests using a session bean to encapsulate the query and return a collection of ValueObjects, thus making the request a single transfer each way.
- The Value Object Assembler pattern uses a Session EJB to aggregate all required data as various types of ValueObjects. This pattern is used to satisfy one or more queries a client might need to execute in order to display multiple data types.

http://www.onjava.com/pub/a/onjava/excerpt/wirelessjava_ch5/index3.html
MIDP GUI programming (Page last updated March 2002, Added 2002-03-25, Author Qusay Mahmoud, Publisher OnJava). Tips:

- Applications with high screen performance needs, like games, need finer control over MIDP screens and should use the javax.microedition.lcdui package which provides the low-level API for handling such cases.
- Always check the drawing area dimensions using Canvas.getHeight() and Canvas.getWidth() [so that you don't draw unnecessarily off screen].
- Not all devices support color. Use Display.isColor() and Display.numColors( ) to determine color support and avoid color mapping [overheads].
- Double buffering is possible by using an offscreen Image the size of the screen. Creating the image: `i = Image.createImage(width, height);` Getting the Graphics context for drawing: `i.getGraphics();` Copying to the screen `g.drawImage(i, 0, 0, 0);`
- Check with Canvas.isDoubleBuffered(), and don't double-buffer if the MIDP implementation already does it for you.
- To avoid deadlock paint() should not synchronize on any object already locked when serviceRepaints() is called.
- Entering alphanumeric data through a handheld device can be tedious. If possible, provide a list of choices from which the user can select.

http://www.javaworld.com/javaworld/jw-12-1999/jw-12-performance.html
Article on the cost of casts (Page last updated December 1999, Added 2000-12-20, Author Dennis M. Sosnoski, Publisher JavaWorld). Tips:

- Casting can be detrimental to performance.
- Improve performance by minimizing casting in heavily used code.
- Some casts take nearly as long as a simple object allocation.
- [Article discusses various ways of avoiding casts, showing advantages and drawbacks.]

http://www.javaworld.com/javaworld/jw-02-2000/jw-02-performance.html
Article on Java 2 collections (Page last updated , Added 2000-12-20, Author Dennis M. Sosnoski, Publisher JavaWorld). Tips:

- Convert collections into arrays for improved access speed.
- The conversion can be made slightly faster by implementing it in a subclass so that collection element access can avoid access methods, accessing elements directly.
- Customized implementations of Hashtables can perform better.

- Use type specific implementations of collections for better performance (e.g. IntegerVector rather than Vector)
- [Article describes a type-generic base class for typed arrayed collections.]

http://www-106.ibm.com/developerworks/webservices/library/ws-testsoap/
Scaling SOAP-based web services. (Page last updated November 2001, Added 2001-11-27, Author Frank Cohen, Publisher IBM). Tips:

- Cache the web services description language (WSDL) in a centralized database and periodically check for newer versions.
- Cache schema definitions for scalability.
- Use simple SOAP data types (String, Int, Float, NegativeInteger).
- Each new data type introduces a serializer to convert from the XML value into a Java value and back again, which may cause performance problems.
- SOAP messages move much more data than the average HTTP GET or POST call, adversely impacting network performance.
- Transactional SOAP calls need to cache the state of sessions.
- [Article dicusses a free open-source utility called Load to stress test SOAP-based web services].

http://www7b.boulder.ibm.com/wsdd/library/techarticles/0106_brown/sessionfacades.html
Rules and Patterns for Session Facades (Page last updated June 2001, Added 2001-07-20, Author Kyle Brown, Publisher IBM). Tips:

- Use the Facade pattern, and specifically Value objects, to transfer all the subset of data needed from an entity bean in one transfer.

http://www.onjava.com/pub/a/onjava/2001/12/19/eejbs.html
EJBs are wonderful (Page last updated December 2001, Added 2001-12-26, Author Tyler Jewell, Publisher OnJava). Tips:

- The out-of-the-box configuration for Entity EJB engines, such as WebLogic, are designed to handle read-write transactional data with the best possible performance.
- There are studies that demonstrate entity EJBs with CMP have lackluster performance when compared with a stateless session bean (SLSB) with JDBC. [Author points out however that SLSB/JDBC combination is less robust, less configurable, and less maintainable].
- Configure separate deployments for each entity bean for different usage patterns (e.g. typical 85% read-only, 10% read-write, 5% batch update), and partition the presentation layer to use the appropriate corresponding deployment (e.g. read requests use the read-only deployment).

http://www.fawcette.com/javapro/2001_12/magazine/features/kkothapalli/
EJB performance tips (Page last updated December 2001, Added 2001-12-26, Author Krishna Kothapalli and Raghava Kothapalli, Publisher JavaPro). Tips:

- Design coarse-grained EJB remote interfaces to reduce the number of network calls required.
- Combine remote method calls into one call, and combine the data required for the calls into one transfer.
- Reduce the number of JNDI lookups: cache the home handles.
- Use session bean wrapper for returning multiple data rows from an entity bean, rather than returning one row at a time.

- Use session beans for database batch operations, entity beans typically operate only one row at a time.
- Use container-managed persistence (CMP) rather than bean-managed persistence (BMP).
- Use entity beans when only a few rows are required for the entity, and when rows need to be frequently updated.
- Use the lowest impact isolation (transaction) level consistent with maintaining data coherency. Highest impact down: TRANSACTION_SERIALIZABLE, TRANSACTION_REPEATABLE_READ, TRANSACTION_READ_COMMITED, TRANSACTION_READ_UNCOMMITED.
- Correctly simulate the production environment to tune the application, and use profiling and other monitroing tools to identify bottlenecks.
- Tune the underlying system, e.g. TCP/IP parameters, file limits, connection pool parameters, EJB pools sizes, thread counts, number of JVMs, JVM heap size, shared pool sizes, buffer sizes, indexes, SQL queries, keep/alive parameters, connection backlogs.
- Use clustering to meet higher loads or consider upgrading the hardware.

http://www-1.ibm.com/servers/eserver/zseries/software/java/perform.html
Both Java specific and Java on OS/390 tips. (Page last updated 2000, Added 2000-10-23, Author ?, Publisher IBM). Tips:

- Use the latest release of the SDK.
- Use zip and jar files. Partition classes to different zip/jar files according to usage.
- Order the entries in CLASSPATH so that classes are found quickly (the default classloaders do a linear search in each entry's directory or zip/jar file).
- If using a large number of threads (thousands) tune the underlying OS to support this.
- Vary the -ss parameter to optimize for threads. Threads generally don't need more than 256K stack size.
- Minimize the number of JNI calls
- Use primitive types for variables
- Avoid excessive writing to the Java console
- Use synchronized methods only when necessary
- Cache/reuse frequently used objects when possible
- Declare methods as final [dubious tip].
- Use static final when creating constants
- Use int instead of long.
- Use local variables in preference to class and instance variables.
- Use arrays instead of vectors
- Consider when to use a temporary variable to manipulate class and instance variables in loops (speed vs. memory)
- Add and delete items from the end of a vector
- Avoid unnecessary cast and instanceof
- Avoid using String when doing a lot of character manipulation. Use StringBuffer instead.
- Avoid using long divides.

http://www.javaworld.com/javatips/jw-javatip26.html
Javaworld tip article, detailing a buffered RandomAccessFile class. (Page last updated 1998?, Added 2000-10-23, Author Nick Zhang). Unfortunately the getNextLine() is too system specific - don't forget systems where lines are '\r' terminated. Tips:

- Use buffered i/o classes.
- Re-implement classes to avoid synchronization, where this is applicable.

http://www.javaworld.com/jw-09-2001/jw-0907-rmi.html
RMI performance tuning (Page last updated September 2001, Added 2001-10-22, Author Ashok Mathew and Mark Roulo, Publisher JavaWorld). Tips:

- Use netperf to measure network bandwidth.
- Consider altering the TcpWindowSize parameter.
- Configure RMI garbage collection by setting the properties `sun.rmi.dgc.client.gcInterval` and `sun.rmi.dgc.server.gcInterval`.
- Send groups of objects together rather than one object at a time.
- Implementing `Externalize` can speed up transfers.
- Pack data to reduce the number and amount of reads and writes, and the amount of data transferred.
- Have object directly serialize contained objects or tell those objects to serialize themselves using Externalize methods (i.e. chain Externalize methods for all contained objects).
- Use special codes to handle special cases such as singleton or reusable objects.
- Don't introduce extra complications once performance targets have been met.

http://www.sys-con.com/java/article.cfm?id=1160
Local entity beans (Page last updated October 2001, Added 2001-10-22, Author Alex Pestrikov, Publisher Java Developers Journal). Tips:

- Local entity beans do not need to be marshalled, and do not incur any marshalling overhead for method calls either: parameters are passed by reference.
- Local entity beans are an optimization for beans which it is known will be on the same JVM with their callers.
- Facade objects (wrappers) allow local entity beans to be called remotely. This pattern incurs very little overhead for remote calls, while at the same time optimizing local calls between local beans which can use local calls.

http://www.developer.ibm.com/library/articles/programmer/haggar_bytecode.html
Nice article on understanding bytecodes by Peter Haggar, Added 2000-10-23, Author of "Practical Java". Explains why a synchronized method is faster than a functionally identical synchronized block. (Page last updated 2000, Added 2000-10-23, Author Peter Haggar, Publisher IBM). Tips:

- Understanding bytecodes can help determine how to improve performance. It can also help you to create smaller sized class files.
- Don't synchronize code unless synchronization is required.
- Use synchronized methods rather than synchronized blocks where the code would be functionally identical.
- The -O option in SDK 1.2 does nothing.
- Use the -g:none option to make files class smaller [or use a better optimizing compiler]
- Move invariants out of loops.
- Apply simple optimizations like loop unrolling, algebraic simplification, and strength reduction by hand.

http://developer.java.sun.com/developer/Books/programming/performance/eperformance/eJavaCh01.pdf
Chapter 1 of "Enterprise Java Performance", "Performance in General". Includes the infamous sentences "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." NO NO NO! If the code meets the performance requirements, DON'T CHANGE IT. Next time guys, ask me to review

your book before you publish. (Page last updated 2000, Added 2000-10-23, Authors Steven Halter & Steven Munroe, Publisher Sun). Tips:

- The simplest code usually performs best.
- Consider performance requirements before coding.
- Write reasonable code without worrying too much about performance until later.
- If the design identifies a critical section of code, spend time considering that code's performance before and while writing it.
- Define performance requirements explicitly. Redefine fuzzy requirements to be more explicit.
- The target machine affects performance requirements.
- Use the simplest classes possible--but no simpler. [Don'cha just love it. And here I was using classes simpler than is possible. Gotta make 'em more complex now.]
- Don?t recalculate things that are constant in loops.
- Reuse objects where possible.
- Choose the correct collection for performance.
- Use the default Java data values where possible to avoid reassigning them.
- Use raw arrays in preference to collections.
- Performance fixing is iterative. Fixing one bottleneck often reveals another previously hidden one.
- Keep performance in mind during the design phase.
- Avoid monopolizing shared resources.
- Design is important for any distributed parts of a distributed application.
- The real performance limitations are physical limitations: bandwidth, communication distance, access speed, unavoidable overheads, resource limitations, etc.
- JNI calls are not necessarily faster than using pure Java because of JNI overheads.
- Schema mapping is complex. [Buy a product that does it for you.]

http://developer.java.sun.com/developer/Books/programming/performance/eperformance/eJavaCh04.pdf
Chapter 4 of "Enterprise Java Performance", "Local/Remote Issues". (Page last updated 2000, Added 2000-10-23, Authors Steven Halter & Steven Munroe, Publisher Sun). Tips:

- RMI over IIOP has a higher overhead than plain RMI.
- Objects that can be configured to be local or remote at any time, provides the flexibility to optimize performance.
- Large grained remote calls [i.e. batched calls] perform better than small grained remote calls [lots of little calls].
- Persistency adds overheads that make persistent objects slower.
- Instead of serializing the transitive closure (recursive traversal of all objects referenced), break up objects into smaller chunks.
- Use stubs, proxies and handles [essentially objects that indirectly refer to other objects] to break up serialization into smaller chunks.
- Unless the application is put together with care, the remote method call costs may dominate.
- Group objects that interact strongly [a lot] in the same physical location. The closer they are, the more efficient their interaction.
- Cache in the client any read-only objects, for the whole session. Replicate any data needed so that queries run locally in the client.
- Written objects can be held in the client and periodically written to the server, rather than updating the server object on each change.

- Good partitioning of objects in distributed applications limits interactions between objects in different partitions and takes advantage of local method access for objects within each partition.
- Application partitioning is best addressed early in the design.

http://developer.java.sun.com/developer/JDCTechTips/2001/tt0327.html
How to use java.rmi.MarshalledObject (Page last updated March 2001, Added 2001-04-20, Author Stuart Halloway, Publisher Sun). Tips:

- MarshalledObject lets you postpone deserializing objects. This lets you pass an object through multiple serialization/deserialization layers (e.g. passing an object through many JVMs), without incurring the serialization/deserialization overheads until absolutely necessary.

http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html
Doug Bell's article with various low-level techniques and benchmark applets. (Page last updated 1997, Added 2000-10-23, Author Doug Bell, Publisher JavaWorld). Tips:

- Don't optimize unless necessary. Optimizing can: introduce new bugs; make code harder to understand and maintain; reduce the extensibility of the code.
- 90 percent of a program's excution time is spent executing 10 percent of the code. (Some people use the 80 percent/20 percent rule). Optimizing the other 90 percent of the program (where 10 percent of the execution time was spent) has no noticeable effect on performance.
- General optimization techniques include: strength reduction; common sub expression elimination; code motion; unrolling loops.
- Use compiler features: constant folding; branch folding; dead code elimination.
- Use the -O option (javac -O).
- Use a profiler to identify bottlenecks.
- Always time the code before and after making changes to verify that, at least on the test platform, your changes improved the program.
- Try to make each timing test under identical conditions.
- If possible, contrive a test that doesn't rely on any user input, as the variations in user response can cause the results to fluctuate.

http://www-106.ibm.com/developerworks/ibm/library/i-tuning/?open
Tuning the IBM JVM and Linux (Page last updated May 2001, Added 2001-10-22, Authors Duc Vianney and James Phelan, Publisher IBM). Tips:

- [Article also has detailed coverage of tuning Linux].
- Use the -Xms and -Xmx parameters to set the heap size.
- Use -verbosegc to measure garbage collection statistics.
- Keep heap size smaller than physical memory.
- Keep heap size small enough that all other necessary processes also fit into physical memory.
- The IBM JVM has extra options to control JVM< heap size expansion and shrinkage.
- Use -Xrunhprof to profile the application [article gives and example of using -Xrunhprof to tune].
- Use local variables where possible.
- Use int instead of long.
- Use arrays instead of vectors.
- Use primitive types such as int and double instead of objects.
- Use exceptions only when necessary.

- Reuse objects as much as possible.
- Avoid writing to the console.
- Cache frequently used objects whenever possible.
- Declare methods as final. Classes and methods that aren't going to be redefined should be declared as final.
- Declare constants as static final.
- Limit the use of synchronized methods.
- Null old object references.
- Cache with soft references.
- Cache information that will be reused and is expensive to generate.
- Use jar files.

http://www.awlonline.com/product/0,2627,0201616467,00.html
Peter Haggar's Practical Java Programming Language Guide. (Page last updated 2000, Added 2001-01-19, Author Peter Haggar, Publisher Addison-Wesley). Tips:

- Focus initially on design, data structures, and algorithms.
- Do not rely on compile-time code optimization.
- Understand runtime code optimization.
- Use StringBuffer, rather than String, for concatenation.
- Minimize the cost of object creation.
- Guard against unused objects.
- Minimize synchronization.
- Use stack variables whenever possible.
- Use static, final, and private methods to allow inlining.
- Initialize instance variables only once.
- Use primitive types for faster and smaller code.
- Do not use an Enumeration or an Iterator to traverse a Vector.
- Use System.arraycopy for copying arrays.
- Prefer an array to a Vector or ArrayList.
- Reuse objects whenever possible.
- Use lazy evaluation.
- Optimize source code by hand.
- Compile to native code.

http://www.onjava.com/pub/a/onjava/synd/2001/08/15/embedded.html
Performance tuning embedded Java (Page last updated August 2001, Added 2001-08-20, Author Vincent Perrier, Publisher OnJava). Tips:

- All the following affect embedded Java performance: hardware processor selection; (real-time) operating system selection; supported Java APIs; application reliability and scalability; graphics support; and the ability to put the application code into ROM.
- Various approaches for boosting bytecode execution speed include: a JIT compiler (usually too big for embedded systems); an ahead-of-time compiler (requires more ROM, may disallow or slowdown dynamically loaded classes); a dynamic adaptive compiler (half-way house between last two options); putting the Java application code into ROM; rewriting the JVM interpretation loop in assembly; using a Java hardware accelerator.
- Use the lightweight graphical toolkit.
- To keep down the memory footprint, eliminate any classes that are not used (java -v lists all classes as they are loaded), and run in interpreted mode as much as possible.
- Benchmark results are not necessarily applicable to your application [article reviews the applicability of standard and proprietary benchmarks].

http://www.javaworld.com/javaworld/jw-07-2001/jw-0720-cache.html
Caching (Page last updated July 2001, Added 2001-08-20, Author Jonathan Lurie, Publisher JavaWorld). Tips:

- Nice description of caching using a filing system analogy.
- Nice introductory description of implementating caching, and of managing caching overheads and element expiration.

http://www.sys-con.com/java/article.cfm?id=712
J2EE challenges (Page last updated June 2001, Added 2001-07-20, Author Chris Kampmeier, Publisher Java Developers Journal). Tips:

- Thoroughly test any framework in a production-like environment to ensure that stability and performance requirements are met.
- Each component should be thoroughly reviewed and tested for its performance and security characteristics.
- Using the underlying EJB container to manage complex aspects such as transactions, security, and remote communication comes with the price of additional processing overhead.
- To ensure good performance use experienced J2EE builders and use proven design patterns.
- Consider the impact of session size on performance.
- Avoid the following common mistakes: Failure to close JDBC result sets, statements, and connections; Failure to remove unused stateful session beans; Failure to invalidate HttpSession.
- Performance test various options, for example, test both Type 2 and Type 4 JDBC drivers; Use a load-generation tool to simulate moderate loads; monitor the server to identify resource utlization.
- Perform code analysis and profiling.
- Performance requirements include: the required response times for end users; the perceived steady state and peak user loads; the average and peak amount of data transferred per Web request; the expected growth in user load over the next 12 months.
- Note that peak user loads are the number of concurrent sessions being managed by the application server, not the number of possible users using the system.
- Larger loads require greater amounts of hardware to satisfy that load.
- Applications that perform very little work can typically handle many users for a given amount of hardware, but can scale poorly as they spend a large percentage of time waiting for shared resources.
- Applications that perform a great number of computations tend to require much more hardware per user, but can scale much better than those performing a small number of computations.
- Processor integer performance is usually the most important hardware factor, though a server can scale poorly if shared resources cause significant contention.
- Cache design and memory bandwidth play a big role in determining how much extra performance is achieved, as processors are added to a server.
- Additional capacity should be designed into the system.
- Extrapolate from known performance test results to predict the performance of the system when varying amount of resources are available.

http://dev2dev.bea.com/articlesnews/discussion/thread.jsp?forum=1&thread=33
EJB Clustering (Page last updated February 2002, Added 2002-04-26, Author Tyler Jewell, Publisher BEA). Tips:

- Four locations that can provide clustering logic for an EJB are: the JNDI naming server where the home stub is bound, the container, the home stub, and the remote stub.

http://www.sys-con.com/java/article.cfm?id=673
J2EE Application servers (Page last updated April 2001, Added 2001-04-20, Authors Christopher G. Chelliah and Sudhakar Ramakrishnan, Publisher Java Developers Journal). Tips:

- A scalable server application probably needs to be balanced across multiple JVMs (possibly pseudo-JVMs, i.e. multiple logical JVMs running in the same process).
- Performance of an application server hinges on caching, load balancing, fault tolerance, and clustering.
- Application server caching should include web-page caches and data access caches. Other caches include caching servers which "guard" the application server, intercepting requests and either returning those that do not need to go to the server, or rejecting or delaying those that may overload the app server.
- Application servers should use connection pooling and database caching to minimize connection overheads and round-trips.
- Load balancing mechanisms include: round-robin DNS (alternating different IP-addresses assigned to a server name); and re-routing mechanisms to distribute requests across multiple servers. By maintaining multiple re-routing servers and a client connection mechanism that automatically checks for an available re-routing server, fault tolerance is added.
- Using one thread per user can become a bottleneck if there are a large number of concurrent users.
- Distributed components should consider the proximity of components to their data (i.e., avoid network round-trips) and how to distribute any resource bottlenecks (i.e., CPU, memory, I/O) across the different nodes.

http://java.oreilly.com/news/jsptips_1100.html
Hans Bergsten's top ten JSP tips (Page last updated November 2000, Added 2001-01-19, Author Hans Bergsten, Publisher O'Reilly). Tips:

- The include *directive* (<%@ include file="filename.inc" %>) is faster than the include *action* (<jsp:include page="pagename.jsp" flush="true"/>).
- *redirect*s are slower than *forward*s because the browser has to make a new request.
- Database access is typically very expensive in terms of server resources. Use a connection pool to share database connections efficiently between all requests, but don't use the JDBC ResultSet object itself as the cache object.

http://www.javaworld.com/javaworld/jw-07-2001/jw-0713-optimism.html
The Optimistic Locking pattern (Page last updated July 2001, Added 2001-07-20, Author Yasmin Akbar-Husain and Eoin Lane, Publisher JavaWorld). Tips:

- Pessimistic locking, where database data is locked when read, can lead to high lock contention.
- Optimistic locking only checks data integrity at update time, so has no lock contention [but can have high rollback costs]. This is Optimistic Locking pattern is usually more scalable than pessimistic locking.
- Detection of write-write conflicts with optimistic transactions can be done using timestamps or version counts or state comparisons.

http://www.microjava.com/articles/techtalk/recycle
Reusing objects in embedded Java (Page last updated July 2001, Added 2001-07-20, Author Angus Muir and Roman Bialach, Publisher Micro Java). Tips:

- A lot of object creation and destruction can lead to a fragmented heap, which reduces the ability to create further objects.
- Define the bulk of memory you need (buffers, etc.) up-front at initialization, and use object pooling to avoid further creation or destruction of objects.
- Throwing/catching exceptions are tremendously expensive.
- Pooling is not always faster than object creation.

http://www.microjava.com/articles/techtalk/recylcle2
Object recycling part 2 (Page last updated February 2002, Added 2002-02-22, Author Angus Muir and Roman Bialach, Publisher Micro Java). Tips:

- The efficiency of pooling objects compared to creating and disposing of objects is highly dependent on the size and complexity of the objects.
- Object pools have deterministic access and reclamation costs for both CPU and memory, whereas object creation and garbage collection can be less deterministic.

http://www.microjava.com/chapter2
Chapter 2, "Java: Fat and Slow?", of "Java 2 Micro Edition: Professional Developer's Guide" referenced from http://www.microjava.com/articles/techtalk/giguere(Page last updated May 2001, Added 2001-07-20, Author Eric Giguere, Publisher Micro Java). Tips:

- Reduce compiled code size by using implicit instruction bytcodes wherever possible. For example, limiting a method to four or fewer local variables (three on non-static methods as "this" takes the first slot), allows the compiler to use implicit forms of instructions (such as aload, iload, fload, astore, istore, fstore, and so on).
- Similarly numbers -1, 0, 1, 2, 3, 4 ,5 have special bytecodes
- Java class files are standalone - no data is shared between class files. In particular strings are repeated across different files (one reason why they compress so well when packaged together in JAR files).
- An empty class compiles to about 200 bytes, of which only 5 bytes are bytecode.
- There are no instructions for initializing complete arrays in the Java VM. Instead, compilers must generate a series of bytecodes that initialize the array element by element. This can make array initialization slow, and adds bytecode to the class.
- You can reduce bytecode bloat from array initialization by encoded values in strings and using those strings initialize the arrays.
- Explicitly set references to null when they are no longer needed to ensure that the objects can be garbage collected.
- Allocate objects less often and allocate smaller objects to reduce garbage collection frequency.

http://www.javaworld.com/javaworld/jw-03-1996/jw-03-animation.html
Animation in java applets article. Old article, but basically sound (the basics haven't changed). (Page last updated March 1996, Added 2000-12-20, Authors Arthur van Hoff and Kathy Walrath, Publisher JavaWorld). Tips:

- Use a separate thread to draw the animation. Do not use the paint() method.
- Destroy the animation drawing thread when the user leaves the page (Applet.stop() is called) to avoid consuming CPU when nothing is being viewed.

- Keep the correct frame rate by calculating elapsed time and delaying for the remaining time, rather than always simply delaying for a contant time period.
- Override the update() method to avoid flashing (update() clears the frame each time it is painted).
- Use double buffering to eliminate further flashing and usually faster drawing: drawing offscreen is potentially faster, and mapping blocks of pixels oto the screen is normally very fast.
- Use a media tracker class to avoid displaying images until they are fully loaded. [This still applies, but Java 2 has more interfaces to help you do this.]
- Use image strips to load multiple images in one action.
- Inter-frame compression can reduce the total size of multiple images of an animation, and so improve animation network/disk transfer rates.

http://java.sun.com/docs/books/tutorial/uiswing/painting/animation.html
Another tutorial from Sun. This ones on animation (Page last updated ?, Added 2000-12-20, Author ?, Publisher Sun). Tips:

- Normal frame rates for animation: 8 frames per second (fps) for poor quality animation; 12 fps for standard animation; 24 fps for short bursts of smooth, realistic motion.
- Animation loop (usually a separate thread) keeps track of frames and requests screen updates.
- Suspend the animation whenever it is not visible.
- Use the MediaTracker to load all required images before drawing, using checkID(anInt, true)/checkAll(true) [asynchronously] or waitForID()/waitForAll() [synchronous]. [example code included in article]
- Combine images in a single file (e.g. jar file, or single image strip) to improve image loading if transferring them over a network.

http://www.cs.tcd.ie/courses/baict/bacs/sf/Animation/
Basic animation tutorial (Page last updated ?, Added 2000-12-20, Author Dave ?, Publisher ?). Tips:

- Avoid flicker by overriding the update() method to avoid blanking the canvas.
- Use a separate thread to manage the calculations and drawing.
- Use MediaTracker to load all required images before drawing.
- Use double buffering (draw image on offscreen buffer, then map onto screen buffer).
- Use synchronization to synchronize methods in the two threads.

http://www.sys-con.com/java/article.cfm?id=713
Moving from JSP to EJB (Page last updated June 2001, Added 2001-06-18, Author Patrick Sean Neville, Publisher Java Developers Journal). Tips:

- Entity EJBs should contain aggregate get/set methods that return chunks of data rather than fine-grained get/set methods for individual attributes, to reduce unnecessary database, transactional, and network communication overheads.
- Avoid stateful session beans as they are resource-heavy, since one instance is maintained for each client.
- Under heavy loads, entity beans should do more than merely represent a table in a database. If you are merely retrieving and updating data values, consider using JDBC within session beans instead.

- If you have one large database host but only a small Web and middleware host, consider moving much of your logic into stored procedures and calling them via JDBC in session beans.
- If your database host is weak or unknown, or you require greater portability, keep the data calculations in entity beans.
- Consider using a single stateless session bean to provide access to other EJBs (this is a façade pattern). This optimizes multiple EJB references and calls by keeping them in-process.
- Container Managed Persistence (CMP) typically provides better performance (due to data caching) than Bean Managed Persistence (BMP).

http://www.devx.com/judgingjava/articles/maso/default.asp
Judging various aspects of Java, including performance (Page last updated May 2001, Added 2001-06-18, Author Brian Maso, Publisher DevX). Tips:

- J2EE defines component models with high scalability potential. Maximizing scalability requires sticking to stateless session beans and handling all database interactions programmatically (through pooled JDBC connections).
- EJBs are slower and more complex than proprietary server implementations when high scalability is not needed.
- Java (to 1.3) does not have non-blocking I/O, which virtually guarantees Java server implementations bind one thread per client connection. This limits communication throughput. Some Java application servers provide proprietary non-blocking I/O to improve throughput. From the 1.4 SDK, Java includes non-blocking I/O.

http://developer.java.sun.com/developer/technicalArticles/JavaLP/Interposing/
Using java.lang.reflect.Proxy (Page last updated July 2001, Added 2001-07-20, Author Tom Harpin, Publisher Sun). Tips:

- Interposing proxy objects is a useful approach to trace or profile method calls.
- The java.lang.reflect.Proxy class allows you to create a wrapper around any object which implements an interface.

http://developer.java.sun.com/developer/Books/performance/performance2/appendixa.pdf
Appendix A (Garbage Collection) of "Java Platform Performance: Strategies and Tactics." (Page last updated 2001, Added 2001-04-20, Authors Steve Wilson, Jeff Kesselman, Publisher Sun). Tips:

- Large RAM requirements can force the OS to use virtual memory, which slows down the application.
- Most JVM implementations will not dereference temporary objects until the method has gone out of scope, even if the object is created in an inner block which has gone out of scope. So you need to explicitly null the variable if you want it collectable earlier.
- Adding a finalizer method extends the life of the object, since it cannot be collected until the finalize() method is run.
- Do not use finalizers to free resources in a timely manner.

http://developer.java.sun.com/developer/technicalArticles/InnerWorkings/JDCPerformTips/index.html
Various performance tips from a JavaOne 1998 presentation. (Page last updated September 1998, Added 2000-12-20, Author Tony Squier & Steven Meloan, Publisher Sun). Tips:

- Minimize the number of times that an applet has to request data from the server.

- Package Applet images into a single class file.
- Use Thread pools where these improve performance.
- Use BufferedIO streams to access URLConnection's Input/Output streams.

http://www.onjava.com/pub/a/onjava/2001/09/26/load.html
Load Balancing Web Applications (Page last updated September 2001, Added 2001-10-22, Author Vivek Veek, Publisher OnJava). Tips:

- DNS round-robin sends each subsequent DNS lookup request to the next entry for that server name. This provides a simple machine-level load-balancing mechanism, but is only appropriate for session independent or shared-session servers.
- DNS round-robin has no server load measuring mechanisms, so requests can still go to overloaded servers, i.e. the load balancing can be very unbalanced.
- Hardware load-balancers solve many of the problems of DNS round-robin, but introduce a single point of failure.
- A web server proxy can also provide load-balancing by redirecting requests to multiple backend webservers.

http://www.javaworld.com/javaworld/jw-02-2001/jw-0202-cachedrow.html
Article on using CachedRowSet, a ResultSet that doesn't need continuous connection to the database (Page last updated February 2001, Added 2001-02-21, Author Taylor G. Cowan, Publisher JavaWorld). Tips:

- CachedRowSet provides cached result sets that do not require continuous connection to the database, allowing connections to be reused more efficiently.
- Using CachedRowSet lets you batch updates, and execute them asynchronously.
- CachedRowSet also supports offline work which is later synchronized.
- CachedRowSet is probably not appropriate for managing large datasets.

http://www.devx.com/premier/mgznarch/Javapro/2002/03mar02/kj0302/kj0302-1.asp
JMS vs RMI (Page last updated February 2002, Added 2002-02-22, Author Kevin Jones, Publisher DevX). Tips:

- RMI calls marshall and demarshall parameters, adding major overhead.
- Every network communication has several overheads: the distance between the sender and the receiver adds a minimum latency (limited by the speed the signal can travel along the wire, about two-thirds of the speed of light: London to New York would take about 3 milliseconds); each network router and switch adds time to respond to data, on the order of 0.1 milliseconds per device per packet.
- Part of most network communications consists of small control packets, adding significant overhead.
- One RMI call does not generally cause a noticeable delay, but even tens of RMI calls can be noticeable to the users.
- Beans written with many getXXX() and setXXX() methods can incur an RMI round trip for every data attribute.
- Messaging is naturally asynchronous, and allows an application to decouple network communications from ongoing processing, potentially avoiding threads from being blocked on communications.

http://www.sys-con.com/java/article.cfm?id=1308
Proxy code generation (Page last updated February 2002, Added 2002-02-22, Author Paul McLachlan, Publisher Java Developers Journal). Tips:

- Generative programming is a class of techniques that allows for more flexible designs without the performance overhead often encountered when following a more traditional programming style. JSP engines are one example. java.lang.reflect.Proxy is another.
- More advanced code obfuscations (such as control-flow obfuscation) can produce slower programs as the obfuscated bytecode is more difficult to optimize by the JIT or HotSpot compiler.
- A reflective lookup [obtaining the method reference from its name] is much slower than a reflective invoke [invoking the method from the reference] once you have a method reference.
- [Article provides an implementation of the JNI call using the JVM_OnLoad() function to trap class bytecodes as they are loaded].
- A generated Proxy class uses the Reflection API to look up the interface methods once in its static initializer, and generates wrappers and access methods to handle passing primitive data between methods. [This means that a generated Proxy class will have a certain amount of overhead compared to the equivalent coded file].

http://www-106.ibm.com/developerworks/java/library/j-jtctips/j-jtc0319a.html
Finalizers (Page last updated March 2002, Added 2002-04-26, Author Phil Vickers, Publisher IBM). Tips:

- Adding finalizers to your code makes GC much more expensive and unpredictable.
- Finalizers are not executed at a predictable time.

http://developer.java.sun.com/developer/community/chat/JavaLive/2001/jl0327.html
Sun community chat session: Tuning the Java Runtime for "Big Iron" (Page last updated March 2001, Added 2001-04-20, Author Edward Ort, Publisher Sun). Tips:

- Use the -server option. Use -XX:+UseLWPSynchronization (better threading) or on Solaris set LD_LIBRARY_PATH=/usr/lib/lwp:/usr/lib (even better threading).
- Set the "young" generation space to 1/4 to 1/3 of heap space, e.g. -Xms1024m -Xmx1024m -XX:NewSize=256m -XX:MaxNewSize=256m. On Solaris use vmstat, pstat (utilities) and -verbose:gc (runtime option).
- GC is single-threaded (at least to 1.3.x), so cannot take advantage of multiple-CPUs (i.e. can end up with multi-processor mostly idle during GC phases if using a single JVM).
- Too many threads can lead to thread "starvation" [presumably thrashing].
- Use at least one thread per CPU, more if any threads will be i/o blocked. On Solaris use the mpstat utility to monitor CPU utlization.
- 1.4 will include concurrent GC that should avoid large GC pauses.
- The biggest performance problem is bad design.
- Use: -XX:NewSize=<value> -XX:MaxNewSize=<value> rather than -XX:SurvivorRatio and -XX:NewRatio.
- Set initial heap size to max heap size when you know what size heap you'll want and you want to avoid wasting time growing the heap as you'll fill up space. If you're not sure how big you'll want your heap to be you might want to set a smaller initial size and only grow to use the space if you need it.
- Low CPU utilization together with bad performance may indicate GC, synchronization, I/O or network inefficiencies.
- -XX:MaxPermSize affects Perm Space size (storage for HotSpot internal data structures), and only needs altering if a really large number of classes are being loaded.
- [The session also discussed some Solaris OS parameters to tune].
- For JDK 1.3, the heap is: TotalHeapSize = -Xmx setting + MaxPermSize; with -Xmx split into new and old spaces [i.e. total heap space is old space + new space + perm space, and

settable heap using -Xmx defines the size of the old+new space. -XX:MaxNewSize defines how much of -Xmx heap space goes to new space].

http://www.sys-con.com/java/article.cfm?id=1301
JMS & JCACHE (Page last updated February 2002, Added 2002-02-22, Author Steve Ross-Talbot, Publisher Java Developers Journal). Tips:

- Asynchronous messaged communications allows subsystems to decouple and work more efficiently in parallel, more closely reflecting actual workflows.
- Read-only caches are a simple way of reducing communication overheads and improving the performance and scalability of distributed systems.
- Event-driven systems tend to be more scalable.
- Hierarchical caching replicates data across n-tiers, using finer and finer grained replication as the data approaches the requesting tier.
- Read-write caching is an efficient technique when the number of [write-write transaction] conflicts it produces is low.

http://www.onjava.com/pub/a/onjava/2001/03/01/pseudo_sessions.html
Pseudo Sessions for JSP, Servlets and HTTP (Page last updated March 2001, Added 2001-03-21, Author Budi Kurniawan, Publisher OnJava). Tips:

- Use pseudo sessions rather than `HttpSessions` to improve web server scalability.
- Pseudo sessions reside on file instead of in memory, thus both decreasing memory and allowing sessions to be distributed across multiple servers.
- Pseudo sessions do not use cookies, instead they alter URLs to encode the session, and so reduce the generation of session objects by cookie-declining browsers.

http://www.javaworld.com/javaworld/jw-02-2001/jw-0223-extremescale.html
Clustering for J2EE and Java application servers. Looks at Bluestone Total-e-server, Sybase Enterprise Application Server, SilverStream Application Server, and WebLogic Application Server. (Page last updated February 2001, Added 2001-03-21, Author Abraham Kang, Publisher JavaWorld). Tips:

- A cluster in this context is a group of machines working together to transparently provide enterprise services.
- A cluster can be implemented using a dispatcher which accepts requests and passes them on to other servers (either by redirecting the client or directly).
- Clusters target to provide scalability and high-availability.
- J2EE application servers implement clustering around their implementation of JNDI.
- Clustering should allow failover if a machine/process crashes. For stateful sessions, this requires state replication.
- Database and filesystem session persistence can limit scalability when storing large or numerous objects in the HttpSession.
- To scale the static portions of your Website, add Web servers; to scale the dynamic portions of your site, add application servers.

http://www.messageq.com/communications_middleware/timberlake_1.html
Multicasting efficiency (Page last updated January 2002, Added 2002-02-22, Author Paul Timberlake, Publisher Message MQ). Tips:

- When dealing with large numbers of active listeners, multicast publish/subscribe is more efficient than broadcast or multiple individual connections (unicast).

- When dealing with large numbers of listeners with only a few active, or if dealing with only a few listeners, multicasting is inefficient. This scenario is common in enterprise application integration (EAI) systems. Inactive listeners require all missed messages to be resent to them in order when the listener becomes active.
- A unicast-based message transport, such as message queuing organized into a hub-and-spoke model, is more efficient than multicast for most application integration (EAI) scenarios.

http://www.fawcette.com/javapro/2002_03/magazine/columns/proshop/default.asp
NIO (Page last updated Daniel F. Savarese, Added 2002-02-22, Author February 2002, Publisher JavaPro). Tips:

- GatheringByteChannel lets you to write a sequence of bytes from multiple buffers, and ScatteringByteChannel allows you to read a sequence of bytes into multiple buffers. Both let you minimize the number of system calls meade by combining operations that might otherwise require multiple system calls.
- Selector allows you to multiplex I/O channels, reducing the number of threads required for efficient concurrent I/O operations.
- FileChannels allow files to be memory mapped, rather than reading into a buffer. This can be more efficient. [But note that both operations bring the file into memory in different ways, so which is faster will be system and data dependent].

http://www.nature.com/Physics/Physics.taf?g=&file=/physics/highlights/6882-2.html&filetype=&_UserReference=C0A804EE46B4E669A2A54F7523EA3CC1CE04
Optimizing Searches via Rare Events (Page last updated April 2002, Added 2002-05-19, Authors ANDREA MONTANARI & RICCARDO ZECCHINA, Publisher Nature). Tips:

- Re-starting a search algorithm at random times can improve the average time required to reach the solution.

http://developer.java.sun.com/developer/technicalArticles/Programming/compression/
Compression in Java (Page last updated February 2002, Added 2002-02-22, Author Qusay H. Mahmoud and Konstantin Kladko, Publisher Sun). Tips:

- Compression techniques have efficiencies that vary depending on the data being compressed. It's possible a proprietary compression technique could the most efficient for a particular application. For example, instead of transmitting a compressed picture, the component objects that describe how to draw the picture may be a much smaller amount of data to transfer.
- ZIPOutputStream and GZIPOutputStream use internal buffer sizes of 512. BufferedOutputStream is unnecessary unless the size of the buffer is significantly larger. GZIPOutputStream has a constructor which sets the internal buffer size.
- Zip entries are not cached when a file is read using ZipInputStream and FileInputStream, but using ZipFile does cache data, so creating more than one ZipFile object on the same file only opens the file once.
- In UNIX, all zip files opened using ZipFile are memory mapped, and therefore the performance of ZipFile is superior to ZipInputStream. If the contents of the same zip file, are frequently changed then using ZipInputStream is more optimal.
- Compressing data on the fly only improves performance when the data being compressed are more than a couple of hundred bytes.

http://www.microjava.com/articles/techtalk/kvmprogramming
Porting to KVM (Page last updated February 2002, Added 2002-02-22, Author Shiuh-Lin Lee, Publisher Micro Java). Tips:

- Minimize program runtime size. Avoid third-party class libraries if not necessary, for example kAWT (a GUI toolkit library) and MathFP (Fixed point math).
- Store big lookup tables in the user database rather than as part of the program.
- Call GC functions manually.
- Dispose of Objects; close the database and the network connections as soon as they are no longer needed.
- Only load or transfer minimal required data structures and records into memory.
- Avoid float and double calculations.
- Avoid data conversions: store and use the data in the final required format, or execute conversions on the server.
- Use client caching.
- Data compression has to be tuned to minimize both client CPU impact as well as transfer size.
- Use tabbed panels to hold different groups of information. Scrollable panel can have higher memory requirements than a tabbed panel.
- Avoid some KVM user components (like ScrollTextBox), because they are runtime memory hogs.
- Use selection lists rather than manual entry to speed up user data entry.

http://www-106.ibm.com/developerworks/library/j-leaks/?dwzone=java
Tracking Memory leaks (Page last updated February 2001, Added 2001-03-21, Author Jim Patrick, Publisher IBM). Tips:

- An object is only counted as being unused when it is no longer referenced. If objects remain referenced unintentionally, this is a memory leak.
- If you get a java.lang.OutOfMemoryError after a while, memory leakage is a strong suspect.
- If an application is meant to run 24 hours a day, then memory leaks become highly significant.
- Most JVMs grow towards the upper heap limit (-Xmx/-mx options) when more memory is required, and do not return memory to the operating system, even if the memory is no longer needed, until the JVM process terminates.
- [Article provides an example of tracking memory leaks using JProbe].

http://developer.java.sun.com/developer/JDCTechTips/2001/tt0807.html
BigDecimal and Enumerations (Page last updated August 2001, Added 2001-08-20, Author Glen McCluskey, Publisher Sun). Tips:

- BigDecimal provides arbitrary-precision floating point number arithmetic, at the cost of performance.
- Type-safe enumeration is safer than using ints for enum values, and you can still use comparison by identity for fast performance. But you lose the performance potential of using the enum values directly as array indices, switch constants and bitmasks.

http://www.webdevelopersjournal.com/columns/connection_pool.html
Article on connection pools (Page last updated September 1999, Added 2001-02-21, Author Hans Bergsten, Publisher Web Developers Journal). Tips:

- Reuse database connections using a connection pool.

- Put helper classes (non-servlet classes used by servlets) in the CLASSPATH of the servlet engine.

http://www.weblogic.com/docs51/techoverview/rmi.html
Weblogic's RMI framework (Page last updated January 1999, Added 2001-03-21, Author , Publisher BEA). Tips:

- Use a single, multiplexed, asynchronous, bidirectional connection for RMI client-to-network traffic instead of the standard reference implementation using multiple sockets.
- Try to improve the serialization mechanism for faster RMI [Externalization is better].
- Use local calls for objects located in the same JVM.
- Minimize distributed garbage collection.
- Use smart stubs which provide data caching and localized execution in addition to the normal remote execution and data fetching capabilities.

ftp://ftp.java.sun.com/docs/j2se/1.4/VolatileImage.pdf and
http://www.javagaming.org/Docs/VolatileImageAPI.htm
Using VolatileImage (Page last updated May 2001, Added 2001-07-20, Author Someone@sun, Publisher Sun). Tips:

- Graphics performance in 1.2 is worse than 1.1. 1.3 is better, and 1.4 should be the fastest yet.
- From 1.2 direct access to image pixels was available, but was too slow to be usable because it involved copying many bits around in memory.
- Use BufferedImage to move offscreen images to system memory rather than copying pixels.
- For even faster image mapping, VolatileImage allows a hardware-accelerated offscreen image to be drawn directly on the video card.
- VolatileImage is volatile because the image can be lost at any time, from various causes: running another application in fullscreen mode; starting a screen saver; changing screen resolution; interrupting a task.
- Only constantly re-rendered images need to be explicitly created as VolatileImage objects to be hardware accelerated. Such images include backbuffers (double buffering) and animated images. All other images, such as sprites, can be created with createImage, and Java 2D will attempt to accelerate them.
- If an image, such as a sprite, is drawn once and copied from many times, Java 2D makes a copy of it in accelerated memory and future copies from the image can perform better.
- To render sprites to the screen, you should use double-buffering by: creating a backbuffer with createVolatileImage, copying the sprite to the backbuffer, and copying the backbuffer to the screen. If content loss occurs, Java 2D re-copies the sprite from software memory to accelerated memory.
- Only some graphics operations (e.g. curved shapes) are accelerated on some platforms. Use profiling to determine what works best for your situation.
- From 1.4 Swing uses VolatileImage for its double buffering.
- VolatileImage.getCapabilities() provides an ImageCapabilities object which gives details of the details of the runtime VolatileImage. The ImageCapabilities allows the application to decide to use less images, images of lower resolution, different rendering algorithms, or various other means to attempt to get better performance from the current situation and platform.

http://developer.java.sun.com/developer/community/chat/JavaLive/2001/jl0131.html
Sun community chat session on "Optimizing Java Program Performance" with Peter Haggar. (Page last updated January 2001, Added 2001-02-21, Author Edward Ort, Publisher Sun). Tips:

- Try faster JVMs if possible.
- Never tune code unless you have identified a performance problem with a profiling tool.
- Beware that tuning techniques may not work well on all platforms.
- Speed start-up time by: minimizing the .class sizes (use -g:none or a shrink tool e.g. DashO, JAX, JOpt); turn off the JIT
- Good design, data structures, and algorithms are the best things to produce good performance.
- Check String manipulation code.
- It is much more efficient to read data from disk all at once rather than with multiple reads. Use the buffered classes when doing i/o.
- `import` is a compile time function, so has no effect on runtime (i.e. import a.b.*; or import a.b.c; make no difference to runtime performance).
- Optimize conditionals to have the most likely true results first.

http://www-106.ibm.com/developerworks/java/library/j-jtp0410/?loc=j
Java transaction management (JTS) (Page last updated April 2002, Added 2002-04-26, Author Brian Goetz, Publisher IBM). Tips:

- A container managing transactions can identify communications to the same database, and automatically convert a two-phase transaction into a more efficient single-phase commit.

http://www.javareport.com/html/from_pages/article.asp?id=249
Article about frameworks and the effective memory management of objects; avoiding memory leaks by design. (Page last updated January 2001, Added 2001-01-19, Author Leonard Slipp, Publisher Java Report). Tips:

- Define the life cycles of objects and the duration of object interrelationships. Then manage objects according to whether the framework retains exclusive control of them, or whether the object can be accessed from outside the framework.
- Minimize the number of objects that can be accessed from outside the framework.
- In general, the creator of an object should be responsible for the objects' life cycle. Where this is not the case, the transfer of ownership of the object should be explicit and emphasized. Similarly object relationship management should be explicit and reversible: for every add() action, there must be a remove(); for every register() action, there must be a deregister().

http://www.javareport.com/html/from_pages/article.asp?id=799&mon=4&yr=2001
Various strategies for connecting to databases (Page last updated March 2001, Added 2001-04-20, Author Prakash Malani, Publisher Java Report). Tips:

- Use pooled connections to reduce connection churn overheads.
- javax.sql.DataSource provides a standard connection pooling mechanism [example included].
- Obtain and release pooled conections within each method that requires the resource if the connection is very short (termed "Quick Catch-and-Release Strategy" in the article). However do not release the connection only to use it again almost immediately, instead hold the connection until it will not be immediately needed.
- The performance penalty of obtaining and releasing connections too frequently is quite small in comparison to potential scalability problems or issues raised because EntityBeans are holding on to the connections for too long.
- The "Quick Catch-and-Release Strategy" is the best default strategy to ensure good performance and scalability.

http://www.java-pro.com/upload/free/features/javapro/2001/01jan01/tm0101/tm0101.asp
Basic article on performance tuning techniques. (Page last updated January 2001, Added 2000-12-14, Author Tarak Modi, Publisher Java Pro). Tips:

- [The compiler concatenates strings where they are fully resolvable, so don't move these concatenations to runtime with StringBuffer.]
- Where the compiler cannot resolve concatenated strings at compile time, the code should be converted to StringBuffer appends, and the StringBuffer should be appropriately sized rather than using the default size.
- Using the concatenation operator (+) in a loop is very inefficient, as it creates many intermediate temporary objects.
- Presizing collections (like Vector) to the expected size is more efficient than using the default size and letting the collection grow.
- Removing elements from a Vector will necessitate copying within the Vector if the element is removed from anywhere other than the end of the collection.
- Cache the size of the collection in a local variable to use in a loop instead of repeatedly calling collection.size().
- Unsynchronized methods are faster than synchronized ones.
- [Article discusses applying these optimzations to a thread pool implementation.]

http://www.numega.com/library/dmpapers/javamem.shtml
Object creation tuning (Page last updated 2000, Added 2001-07-20, Author Daniel F. Savarese, Publisher Numega). Tips:

- Creating and dereferencing too many objects can adversely impact performance.
- Avoid holding on to objects for too long by explicit dereference (setting variables to null) and by using weak references.
- Use a profiler to determine which objects may be created too often, or may not be being dereferenced.
- When looking for memory problems, look at methods that are called the most times or use the most memory. Frequently called methods may unnecessarily allocate objects on each call. Methods that use a lot of memory may not need to use as much memory or they may be a source of memory leaks.
- Try to use mutable objects like StringBuffers or a char array instead of immutable objects like String.
- Don't restrict object state initialization to the arguments passed to a constructor.
- Provide a zero-argument constructor that creates reasonable default values and include setter methods or an init method to allow objects of that class to be reused.
- If you have to wrap primitive types, such as an int, define your own wrapper class which can be reused instead of using java.lang.Integer.
- If you need to create many instances of a wrapper class like Integer, consider writing your algorithm to accept primitive types.
- Use a factory class instead of directly calling the "new" operator, to allow easier reuse of objects.
- Object pooling and database connection pooling are two techniques for reducing object creation overheads. Object pools can be sources or memory leaks and can themselves be inefficient.

http://developer.java.sun.com/developer/technicalArticles/releases/nio/
The java.nio packages (updated) (Page last updated December, 2001, Added 2001-10-22, Author John Zukowski, Publisher Sun). Tips:

- Direct buffers have a higher creation cost than non-direct buffers because they use native system operations rather than JVM operations.
- Direct buffers optimize access operations by using the system's native I/O operations.
- Reduce threads by multiplexing I/O using selectors: The new I/O capabilities, allow you to create a Web server that does not require one thread per connection.

http://www.fawcette.com/javapro/2002_01/magazine/features/rgrehan/
How to Climb a B-tree (Page last updated December 2001, Added 2001-12-26, Author Rick Grehan, Publisher JavaPro). Tips:

- A B-tree outperforms a binary tree when used for external sorting (for example, when the index is stored out on disk) because searching a binary tree cuts the number of keys that need searching in half for every node searched, whereas B-tree searching cuts the number of keys that have to be searched by approximately 1/n, where n is the number of keys on a node.
- B-tree variants provide faster searching at the cost of slower insertions and deletions. Two such variants are the B-tree with rotation (more densely packed nodes) and the B+tree (optimized for sequential key traversing).
- [Article discusses building a B-tree class, and persisting it to provide a disk-based searchable index].

http://www.eweek.com/article/0,3658,s=708&a=23115,00.asp
Database comparison (Page last updated February 2002, Added 2002-04-26, Author Timothy Dyck, Publisher E-Week). Tips:

- SQLServer has driver problems that slow access to it.
- Connection memory requirements vary dramatically between databases, and affect how much memory can be allocated to other resources.
- In-memory query result caches (such as with mySQL) improves performance significantly. (Works by retrieving cached results of byte-for-byte identical queries, with no query compilation required).
- Add extra indexes.
- Arrange the stored order of rows to best satisfy the queries.
- Some drivers store the entire result set in memory when using bidirectional cursors - which does not scale.

http://developer.java.sun.com/developer/JDCTechTips/2002/tt0507.html
File Channels, StackTraceElements, (Page last updated May 2002, Added 2002-05-19, Author Glen McCluskey, Publisher Sun). Tips:

- File Channels (from the 1.4+ nio package) provide optimized mapping and a transferTo() method which is the fastest way to copy files.
- StackTraceElement provides access to the stack from excception objects, useful for analyzing stack elements.

http://www.javaworld.com/javaworld/jw-12-2001/jw-1207-java101.html
Article about garbage collection and finalization. (Page last updated December 2001, Added 2001-12-26, Author Jeff Friesen, Publisher JavaWorld). Tips:

- [No specific performance tips, but its always helpful to know about GC].

http://www-106.ibm.com/developerworks/java/library/j-jtp0305.html
Java Transaction Service (Page last updated March 2002, Added 2002-03-25, Author Brian Goetz, Publisher IBM). Tips:

- Writing every data block to disk when any part of it changes would be bad for system performance. Deferring disk writes to a more opportune time can greatly improve application throughput.
- Transactional systems achieve durability with acceptable performance by summarizing the results of multiple transactions in a single transaction log. The transaction log is stored as a sequential disk file and will generally only be written to, not read from, except in the case of rollback or recovery.
- Writing an update record to a transaction log requires less total data to be written to disk (only the data that has changed needs to be written) and fewer disk seeks.
- Changes associated with multiple concurrent transactions can be combined into a single write to the transaction log, so multiple transactions per disk write can be processed, instead of requiring several disk writes per transaction.

http://www.sys-con.com/weblogic/article.cfm?id=101
HTTP sessions vs. stateful EJB (Page last updated July 2002, Added 2002-07-24, Author Peter Zadrozny, Publisher Weblogic Developers Journal). Tips:

- The comparative costs of storing data in an HTTP session object are roughly the same as storing the same data in a stateful session bean.
- Failure to remove an EJB that should have been removed (from the HTTP session) carries a very high performance price: the EJB will be passivated which is a very expensive operation.

http://java.sun.com/docs/books/performance/1st_edition/html/JPPerformance.fm.html
Chapter 1, "What Is Performance?" of "Java Platform Performance". (Page last updated 2000, Added 2001-11-27, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- Design your software with the target configuration (e.g. RAM) in mind.
- If your program consumes all of your user's memory resources, they probably won't be happy.
- Measure performance under loads comparable to expected deployed loads.
- Perceived performance is a highly important aspect of performance. How fast a program feels is more important than how fast it really is.

http://java.sun.com/docs/books/performance/1st_edition/html/JPProcess.fm.html
Chapter 2, "The Performance Process" of "Java Platform Performance". (Page last updated 2000, Added 2001-11-27, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- It's nearly impossible to achieve good performance through optimizations alone, without considering performance in analysis and design stages.
- Creating clear system and performance requirements is the key to evaluating the success of your project.
- Use cases provide excellent specifications for building benchmarks.
- Specify the limitations of the application: well-defined boundaries on the application scope can provide big optimization opportunities.
- Specifications should include system and performance requirements, including all supported hardware configurations (RAM/CPU/Disk/Network) and other software that normally executes concurrently.

- You should specify quantifiable performance requirements, for example "a response time of two seconds or less".
- Scalability is more dependent on good design decisions than optimal coding techniques.
- Encapsulation leads to slowdowns from increased levels of indirection, but is essential in large, scalable, high-performance systems. For example, using a java.util.List object may be slower than using a raw array, but allows you to change very easily from ArrayList to LinkedList when that is faster.
- Meeting or exceeding your performance requirements should be part of the shipping criteria for your product.
- Once you've determined that a performance problem exists, you need to begin profiling. Profilers are most useful for identifying computational performance and RAM footprint issues.
- Performance tuning is an iterative process. Data gathered during profiling needs to be fed back into the development process.

http://java.sun.com/docs/books/performance/1st_edition/html/JPMeasurement.fm.html
Chapter 3, "Measurement Is Everything" of "Java Platform Performance". (Page last updated 2000, Added 2001-11-27, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- Benchmarks are typically time-related, but can also measure quantities such as how much memory is used.
- A stopwatch is a versatile benchmarking tool.
- System.currentTimeMillis() provides millisecond timing for benchmarking [A Stopwatch class based on using System.currentTimeMillis() is presented].
- Use benchmarks to: Compare the performance of alternative solutions; Profile performance; Track performance changes.
- Micro-benchmarks (repeatable sections of code) can be useful but may not represent real-world behavior. Factors that can skew micro-benchmark performance include Java virtual machine warm-up time, and global code interactions.
- Macro-benchmarks (repeatable test sequences from the user point of view) test your system as actual end users will see it.
- Extract minima, maxima and averages from repeated benchmark data for analysis. Use these to compare progress of benchmarks during tuning. [I like to add the 90th-centile value too].
- Profilers help you find bottlenecks in applications, and should show: the methods called most often; the methods using the largest percentage of time; the methods calling the most-used methods; and the methods allocating a lot of memory.
- The Sun JVM comes with the hprof profiler.
- Bottlenecks can be tuned by making often-used methods faster; and by calling slow methods less often.
- Backtrace methods to understand the context of the bottleneck. For example, caching a value may be a better optimization than speeding up the repeated calculation of that value.
- Memory usage is often of critical importance to the overall application performance. Excessive memory allocation is often one of the first things that an experienced developer looks for when tuning a Java program.
- Examine bottlenecks for memory allocation. For example you may be able to replace a repeated object allocation in a loop with a reusable object allocated once outside the loop.
- Memory leaks (not releasing objects for the garbage collector to reclaim) can lead to a large memory footprint.
- You identify memory leaks by: determining that there is a leak; then identifying the objects that are not being garbage colleted; then tracing the references to those leaking objects to determine what is holding them in memory.

- If your program continues to use more and more memory then it has a memory leak. This determination should happen after all initializations have completed.
- Identify memory leak objects by marking/listing the objects in some known state, then cycling through other states and back to that known state and seeing which extra objects are now present.
- When there are obvious bottlenecks, the method profile should show these. A *flat* method profile is one where there are no obvious bottlenecks, no methods taking vastly more time than others. In this case you should look at cumulative method profiles, which show the relative times taken by a method and all the methods it calls (the *call tree*). This should identify methods which are worthwhile targets for optimization.

http://java.sun.com/docs/books/performance/1st_edition/html/JPIOPerformance.fm.html
Chapter 4, "I/O Performance" of "Java Platform Performance: Strategies and Tactics." (Page last updated 2000, Added 2001-12-27, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- Buffer i/o operations.
- Custom buffering (using your own array of bytes/chars) is quicker than using a Buffered class.
- Application specific i/o can be tuned, e.g. caching in memory frequently served pages of a HTTP server.
- Default Serialization is slow.
- Use the `transient` keyword to define fields to avoid having those fields serialized. Examine serialized objects to determine which fields do not need to be serialized for the application to work.

http://java.sun.com/docs/books/performance/1st_edition/html/JPRAMFootprint.fm.html
Chapter 5, "RAM Footprint" of "Java Platform Performance: Strategies and Tactics." (Page last updated 2000, Added 2001-12-27, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- Virtual memory is many times slower than RAM: try to fit the application into available RAM on the target platform.
- Runtime.totalMemory() and Runtime.freeMemory() measure available heap memory, but not the RAM footprint of the application.
- Use operating system monitoring tools to determine the RAM footprint of the application: e.g. task manager on Windows NT, pmap -x and ps on Solaris.
- Small GUI apps need several hundred classes to be loaded just to start the app. Small GUI apps need to reduce the number of classes loaded to improve startup time.
- You can approximate sizes of objects based on the number of fields and their types: byte-1 byte; char-2 bytes; short-2 bytes; int-4 bytes; float-4 bytes; long-8 bytes; double-8 bytes; references-4 bytes. JVMs will impose additional overheads.
- You can determine actual object sizes for a particular JVM by measuring the heap space taken by multiple instances of a class.
- Use profiling to determine the overal size cost of a class of objects, to determine whether it is worth reducing the size cost of the class.
- Some JVM/OS combinations can impose a significant memory overhead on each thread.
- Use 'java -verbose <MyMainClass>' to identify all classes that are loaded.

http://java.sun.com/docs/books/performance/1st_edition/html/JPClassLoading.fm.html
Chapter 6, "Controlling Class Loading" of "Java Platform Performance: Strategies and Tactics." (Page last updated 2000, Added 2001-12-27, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- To avoid loading unnecessary classes (e.g. when the JIT compiles methods which refer to unused classes), use Class.forName() instead of directly naming the class in source. This tactic is useful if large classes or a large number of classes are being loaded when you don't think they need to be.
- Combine listener functionality into one class to avoid an explosion of generated inner classes. This technique increases maintenance costs.
- Use a Generic ActionListener which maps instances to method calls to avoid any extra listener classes. This has the drawback of losing compile-time checks. java.lang.reflect.Proxy objects can be used to generalize this technique to multiple interfaces.
- Run multiple applications in the same JVM. [Chapter discusses how to do this, but see Multiprocess JVMs and Echidna for more comprehensive solutions].

http://java.sun.com/docs/books/performance/1st_edition/html/JPMutability.fm.html
Chapter 7, "Object Mutability: Strings and other things" of "Java Platform Performance: Strategies and Tactics." (Page last updated 2000, Added 2002-02-22, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- The allocation, initialization, and collection of many short-lived useless objects can cause major inefficiencies in your software, even when running on an advanced runtime such as the HotSpot VM.
- Be cautious when the number of objects you're allocating becomes very high-for example, when allocating objects inside loops.
- For heavy-duty text processing, however, some uses of the String class can become major performance bottlenecks.
- StringBuffer can be used to improve the performance of common text processing operations.
- Avoid creating new strings in compute intensive parts of code. Be careful of the concatenation operators '+' and '+=' when used with strings.
- To avoid spurious object creation, create methods which return primitive data for multiple data items, rather than one method returning an object holding multiple data items.
- Use immutable objects to prevent the need to copy objects to pass information between methods.
- Object pooling small objects is often counterproductive. The overhead of managing the object pool is often greater than the small object penalty. Pooling can also increase a program's memory footprint.
- Pooling large objects (e.g. large bitmaps or arrays) or objects that work with native resources (e.g. Threads or Graphics) can be efficient.

http://java.sun.com/docs/books/performance/1st_edition/html/JPAlgorithms.fm.html
Chapter 8, "Algorithms and data structures" of "Java Platform Performance: Strategies and Tactics." (Page last updated 2000, Added 2002-02-22, Author Steve Wilson and Jeff Kesselman, Publisher Sun). Tips:

- Choosing the best algorithm or data structure for a particular task is one of the keys to writing high-performance software.
- The optimal algorithm for a task is highly dependent on the data and data size.
- Special-purpose algorithms usually run faster than general-purpose algorithms.
- Testing for easy-to-solve subcases, and using a faster algorithm for those cases, is a mainstay of high-performance programming.
- Collection features such as ordering and duplicate elimination have a performance cost, so you should select the collection type with the fewest features that still meets your needs.

- Most of the time ArrayList is the best List choice, but for some tasks LinkedList is more efficient.
- HashSet is much faster than TreeSet.
- Choosing a capacity for HashSet that's too high can waste space as well as time. Set the initial capacity to about twice the size that you expect the Set to grow to.
- The default hash load factor (.75) offers a good trade-off between time and space costs. Higher values decrease the space overhead, but increase the time it takes to look up an entry. (When the number of entries exceeds the product of the load factor and the current capacity, the capacity is doubled).
- Programs pay the costs associated with thread synchronization even when they're used in a single-threaded environment.
- The Collections.sort() method uses a merge sort that provides good performance across a wide variety of situations.
- When dealing with collections of primitives, the overhead of allocating a wrapper for each primitive and then extracting the primitive value from the wrapper each time it's used is quite high. In performance-critical situations, a better solution is to work with plain array structures when you're dealing with collections of primitive types.
- Random number generation can take time. If possible you can pre-generate the random number sequence into an array, and use the elements when required.

http://developer.java.sun.com/developer/Books/performance/ch10.pdf
Chapter 10 (Swing models and renderers) of "Java Platform Performance: Strategies and Tactics." (Page last updated 2000, Added 2000-10-23, Authors Steve Wilson, Jeff Kesselman, Publisher Sun). Tips:

- Swing?s model-view architecture is critical for building scalable programs.
- When changing data stored in models, perform the operations in bulk whenever possible. E.g. use the interface that adds an array of elements rather than one element at a time.
- Use custom models to handle large datasets. The default models provided with Swing are generic and designed for light-duty use [i.e. are slow].
- Custom renderers can sometimes be used to improve performance. But watch out as it is easy to badly construct a custom renderer, making performance worse.
- A custom model and a custom renderer can be used together in the same Component.
- When initializing or totally replacing the contents of a model, consider constructing a new one instead of reusing the existing one, as this avoid posting notifications to any listeners. [Or reuse the object but deregister the listeners first].

http://developer.java.sun.com/developer/qow/archive/134/index.jsp
Improving socket transfer rates (Page last updated May 2001, Added 2001-05-21, Author Rama Roberts, Publisher Sun). Tips:

- The usual "StringBuffer better than String" tip applies to socket communications too.
- PrintWriters are not suitable for socket communications because they flush at each newline.
- Socket.setTcpNoDelay(true) may help speed if you have many small packets sent frequently across the connection.

http://www.theserverside.com/resources/article.jsp?l=Tips-On-Performance-Testing-And-Optimization
Server performance testing (Page last updated 2000, Added 2001-05-21, Author Floyd Marinescu, Publisher The Server Side). Tips:

- Test response times against average current data/user volumes, then repeat the same test against four times as much volume as you expect in 3 years time. This defines your long term target - getting the response times the same for that latter test.
- Response time increasing too much when database is over populated probably indicates lack of or inappropriate indexing on the database.
- Response time increasing exponentially as load increases, you need to improve scalability by optimizing the application or adding resources.
- Use SQL clause with EXPLAIN or similar (e.g. "Explain select * from table where tablefield = somevalue") to ensure that the database is doing an indexed search rather than a linear searches of large datasets.
- Use a profiler to determine object usage, garbage collection behaviour and method bottlenecks in the application.
- Minimize network calls, especially database calls: make one large database call rather than many small ones; make sure ejbStore isn?t storing anything for read only operations; use Details Objects to get entity bean state rather than making many trips for each aspect of state.
- Use caching where possible.
- Use session beans as a façade to your entity beans to encapsulate the workflow of one entire usecase in one network call to one method on a session bean (and one transaction).

http://developer.java.sun.com/developer/technicalArticles/ebeans/sevenrules/
Optimizing entity beans (Page last updated May 2001, Added 2001-05-21, Author Akara Sucharitakul, Publisher Sun). Tips:

- Use container-managed persistence when you can. An efficient container can avoid database writes when no state has changed, and reduce reads by retrieving records at the same time as find() is called.
- Minimize database access in ejbStores. Use a "dirty" flag to avoid writing tee bean unless it has been changed.
- Always cache references obtained from lookups and find calls. Always define these references as instance variables and look them up in the setEntityContext (method setSessionContext for session beans).
- Always prepare your SQL statements.
- Close all database access/update statements properly.
- Avoid deadlocks. Note that the sequence of ejbStore calls is not defined, so the developer has no control over the access/locking sequence to database records.

http://www.sys-con.com/java/article.cfm?id=658
EJB best practices (Page last updated April 2001, Added 2001-05-21, Author Sandra L. Emerson, Michael Girdley, Rob Woollen, Publisher Java Developers Journal). Tips:

- To avoid resources being held unnecessarily for long periods, a transaction should never encompass user input or user think time.
- Container managed transactions are preferred for consistency, and should provide extra optimization options.
- Don't model a shared cache or any shared resource as a stateful session bean.
- Stateless session beans are easier to scale than stateful session beans. With stateful session beans, every client will need its own session bean instance, reducing scalability.
- Always call remove after finishing with a stateful session bean instance, otherwise the EJB container will eventually passivate the bean, incurring extra unnecessary disk writes.

http://www.javaworld.com/javaworld/jw-08-2001/jw-0803-extremescale2.html
J2EE clustering (Page last updated August 2001, Added 2001-08-20, Author Abraham Kang, Publisher JavaWorld). Tips:

- Consider cluster-related and load balancing programming issues from the beginning of the development process.
- Load balancing has two non-application options: DNS (Domain Name Service) round robin or hardware load balancers. [Article discusses the pros and cons].
- To support distributed sessions, make sure: all session referenced objects are serializable; store session state changes in a central repository.
- Try to keep multiple copies of objects to a minimum.

http://www.java-zone.com/free/articles/Kabutz01/Kabutz01-1.asp
SoftReference-based HashMap (Page last updated August 2001, Added 2001-08-20, Author Heinz Kabutz, Publisher Kabutz). Tips:

- WeakHashMaps are not ideal if you want the values to be weakly referenced rather than the keys.
- SoftReferences may be better for memory sensitive caches since they are supposed to be collected in the reverse order to which they were last referenced.
- Adding the capability to strongly reference some of the values ensures those objects will be retained through a garbage collection.

http://forums.itworld.com/webx?14@@.ee6bdcf/598!skip=542
Avoiding memory leaks in EJBs (Page last updated April 2001, Added 2001-05-21, Author Govind Seshadri, Publisher IT World). Tips:

- Make sure that any beans which have session scope implement the HttpSessionBindingListener interface
- Explicitly release any resources that may be used within the bean by implementing the valueUnbound() callback.
- Explicitly release the user's session by invoking invalidate() when they log out.
- Try setting the session invalidation interval to a smaller value than the default 30 minutes.
- Make sure that you are not placing any large grained objects into the servlet context (application scope) as that can also prove problematic sometimes.

http://www-106.ibm.com/developerworks/java/library/j-super.html
Parallel clustering of machines using Java (Page last updated April 2001, Added 2001-04-20, Author Aashish N. Patil, Publisher IBM). Tips:

- [Article describes an implemented architecture for distributing Runnable threads across multiple computer nodes].

http://developer.java.sun.com/developer/TechTips/2000/tt0829.html
The Javap disassembler (Page last updated August 2000, Added 2001-04-20, Author Stuart Halloway, Publisher Sun). Tips:

- [Article describes using the `javap` disassembler, useful for identifying exactly what the code has been compiled into].
- Use the `javap` disassembler to determine the efficiency of generated bytecodes.
- `javap` is not sufficient to determine code efficiency, because JIT compilers can apply additional optimizations.

http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-jfile.html
Speeding up file searching in JFileChooser (Page last updated February 2001, Added 2001-03-21, Author Slav Boleslawski, Publisher JavaWorld). Tips:

- [Article discusses JFileChooser's operation in detail, including multi-threading, filename caching and batched delivery. Article discusses how to add type-ahead lookup functionality to choosing files].

http://www.protomatter.com/nate/java-optimization/
Various tips. (Page last updated 1999?, Added 2000-10-23, Author Nate Sammons, Publisher Sammons). Tips:

- Use StringBuffer for String concatenations, rather than the '+' operator.
- Use static strings, String.intern() or a static Hashtable to reduce the number of occurrances of identical string objects.
- Modify java.lang.String to cache the hashCode if you are using many string keys in hash tables [note Sun added this optimization to the String class in SDK 1.3]
- String.getBytes() is very inefficient with a method call for *every* character. Use getBytes(int, int, byte[] int) instead, or some other mecahnism.
- Use non-synchronized Vector, Hashtable, etc. where possible.
- Size Vector, Hashtable, etc. appropriately.
- Object creation is expensive. Pool your objects where possible.
- Inner class object creation is even more expensive than normal. Use non-public concrete support classes instead.
- Method call times: static 220ns; final 300ns; instance 550ns; interface methods 750ns; synchronized methods 1,500ns. [But times vary enormously depending on the VM and context].
- Use static final methods where possible. [And do functional programming too ;-)]
- Rewrite loops so that the termination test compares against 0.
- Use exception terminated infinite loops for long loops.
- Use System.arraycopy() to copy arrays.
- Use temporary local variables to manipulate instance variables.

http://www.onjava.com/pub/a/onjava/2000/12/15/ejb_clustering.html
"EJB2 clustering with application servers" (Page last updated December 2000, Added 2001-01-19, Author Tyler Jewell, Publisher OnJava). Tips:

- [Article discusses multiple independent ways to load balance EJBs]

http://developer.java.sun.com/developer/community/chat/JavaLive/2000/jl0829.html
Sun community chat session with Steve Wilson and Jeff Kesselman on Java Performance (Page last updated August 2000, Added 2001-01-19, Author Edward Ort, Publisher Sun). Tips:

- HotSpot JVMs can return heap memory to the OS while running.
- HotSpot Client VM (JVM 1.3) is optimized for quick startup time and low-memory footprint. The server VM (HotSpot 1.0/2.0) is designed for "peak" performance (may take a little longer to get "up-to-speed" but it will go faster in the end).
- Always use System.arraycopy to copy arrays.
- Sticky applets available with the 1.3 plugin speeds startup (persistently caches classes on clients). Also put resources together into jar file to reduce download requests.
- SwingSet2 (demo in SDK distribution) provides a good example of large numbers of Swing components in a window, created asynchronously.

- Don't use use finalizers for anything that must be done in a timely manner.
- Use primitives and transients to speed up serialization.
- Use a concentrator object to limit the repaint events to once every 100 milliseconds in heavily loaded systems and in multi-threaded swing environments. There is some overhead for context switching (using invokeLater) into the AWT-event thread, which you want to minimize.
- The key to high performance code is organization and process. Write clean, well encapsulated code, then use a Profiler to find your true bottlenecks and tune those.

http://www.java-pro.com/upload/free/features/Javapro/2001/07jul01/ah0107/ah0107-1.asp
Experiences building a servlet (Page last updated June 2001, Added 2001-06-18, Author Asif Habibullah, Jimmy Xu, Publisher Java Pro). Tips:

- Keep the size of the client tier small so that downloads are fast.
- Use the servlet init() and destroy() methods to start and stop limited and expensive resources, such as database connections.
- Make the servlets thread-safe and use connection pooling.
- Use PreparedStatements rather than plain Statement objects.
- Use database stored procedures.

http://www.javaworld.com/javaworld/javatips/jw-javatip122.html
Typesafe Enumeration gotchas (Page last updated January 2002, Added 2002-01-25, Author Vladimir Roubtsov, Publisher JavaWorld). Tips:

- Implement readResolve() for Serializable Enumeration classes to maintain object identity.
- Ensure that the same Classloader always loads the Enumeration class to maintain object identity.

http://www.javaworld.com/javaworld/jw-01-2002/jw-0104-java101.html
Reference objects (Page last updated January 2002, Added 2002-01-25, Author Jeff Friesen, Publisher JavaWorld). Tips:

- Cache objects such as images in memory for quicker presentation after the first display request.
- Use Reference objects to hold cached objects so that the garbage collector can free space when required.
- [Article discusses in detail the various Reference object types].

http://www.devx.com/upload/free/features/Javapro/2002/03mar02/jt0302/jt0302-1.asp
Optimizing Java for intensive numeric calculations (Page last updated January 2002, Added 2002-02-22, Author James W. Cooper, Publisher DevX). Tips:

- Allocating on the heap (as with object creation) is much slower than allocating on the stack.
- Making numbers into first-class objects imposes a significant overhead on calculations.
- Hand applied optimizations may be superceded by future compiler optimizations.
- Use specialized subtypes to reduce dynamic dispatching.
- Replace objects with their data held and passed as local variables.

http://www-106.ibm.com/developerworks/ibm/library/i-signalhandling/
OS Signal handling in Java (Page last updated January 2002, Added 2002-02-22, Author Chris White, Publisher IBM). Tips:

- [Article describes how to handle operating system signals from within Java. Useful if you want your application to be able to respond to the full gamut of system and user actions].

http://www-106.ibm.com/developerworks/java/library/j-native.html
Natively compiled code from Java source (Page last updated January 2002, Added 2002-02-22, Author Martyn Honeyford, Publisher IBM). Tips:

- Natively compiled code generated from Java source might be faster and might require less memory and disk resources. [But this articles show some JVMs can be faster].
- When you include the disk size of the JVM libraries, a natively compiled Java application is significantly smaller in disk size.
- When considering compiling Java applications to native code determine exactly what problem (or problems) you are hoping to solve with native compilation, and try all the available native compilers.

http://www.javareport.com/html/from_pages/article.asp?id=5769&mon=12&yr=2001
RMI arguments (Page last updated December 2001, Added 2002-02-22, Author Scott Oaks, Publisher Java Report). Tips:

- Some application servers can automatically pass parameters by reference if the communicating EJBs are in the same JVM. To ensure that this does not break the application, write EJB methods so that they don't modify the parameters passed to them.

http://www.fawcette.com/javapro/2002_01/online/online_eprods/j2ee_sspielman1_25/
Choosing an application server (Page last updated January 2002, Added 2002-02-22, Author Sue Spielman, Publisher JavaPro). Tips:

- A large-scale server with lots of traffic should make performance its top priority.
- Performance factors to consider include: connection pooling; types of JDBC drivers; caching features, and their configurability; CMP support.
- Inability to scale with reliable performance means lost customers.
- Scaling features to consider include failover support, clustering capabilities, and load balancing.

http://www.java-zone.com/free/articles/Liotta01/Liotta01-1.asp
Notated keys to access elements of nested Maps. (Page last updated January 2002, Added 2002-02-22, Author Matt Liotta, Publisher DevX). Tips:

- Use dot separated, concatenated strings to optimize access to elements of nested Maps by caching elements in the top level Map.

http://www-106.ibm.com/developerworks/java/library/j-jtctips/j-jtc0117b.html
The Garbage Collector (Page last updated January 2002, Added 2002-01-25, Author Phil Vickers, Publisher IBM). Tips:

- In most current JVMs (prior to 1.4) GC starts off by locking out all other threads in the JVM. GC is a stop-the-world, synchronous operation. Non-generationl GC requires scanning the stacks of every thread and the entire Java heap.
- Calling System.gc() explicitly is not good for performance, as it can be called when GC is not necessary, but will still result in a long pause of all JVM operations.

http://www-106.ibm.com/developerworks/java/library/j-jtctips/j-jtc0117e.html
Object management (Page last updated January 2002, Added 2002-01-25, Author Mark Bluemel, Publisher IBM). Tips:

- Avoid retaining objects accidentally, by holding references beyond an appropriate time for their release.
- Use profiling tools to identify unintentionally retained objects.
- Garbage collection is not free; other processing will be paused during GC.
- Try to reuse objects in preference to discarding and re-creating them.

http://www-106.ibm.com/developerworks/java/library/j-jtctips/j-jtc0117a.html
JViewport scrolling performance (Page last updated January 2002, Added 2002-01-25, Author Heather Brailsford, Publisher IBM). Tips:

- JViewport.BLIT_SCROLL_MODE is the default scrolling mode for JViewport in SDK 1.3 (available since 1.2.2). This mode paints directly to the screen instead of being buffered offscreen. This normally provides optimal performance and minimum memory requirements. However complex images may display some intermediate paint operations if the painting is not fast enough, giving jerky or flashing images. If this is unacceptable, try the alternate modes: setScrollMode(BACKINGSTORE_SCROLL_MODE) (intermediate performance, higher memory requirements); or setScrollMode(JViewport.SIMPLE_SCROLL_MODE) (slowest).

http://www-106.ibm.com/developerworks/java/library/j-jtctips/j-jtc0117d.html
Using JNI Get* calls (Page last updated January 2002, Added 2002-01-25, Author Mark Bluemel, Publisher IBM). Tips:

- If you use JNI Get* calls (for example, GetStringCritical), you must always use the corresponding Release* call (for example, ReleaseStringCritical) when you have finished with the data, even if the isCopy parameter indicates that no copy was taken.

http://www.devx.com/upload/free/Features/Javapro/2002/02feb02/bk0202/bk0202-1.asp
Servlet 2.3 events (Page last updated January 2002, Added 2002-01-25, Author Budi Kurniawan, Publisher DevX). Tips:

- The Servlet 2.3 specification adds application and session events. [Event driven applications can often be scaled more easily than process driven applications].

http://developer.java.sun.com/developer/onlineTraining/webcasts/chicago/pdf/j2se.pdf
Sun presentation on J2SE performance strategies (originally accessed from Reginald Hutcherson's page) (Page last updated May 2001, Added 2001-06-18, Author Reginald Hutcherson, Publisher Sun). Tips:

- The Sun 1.3 JVM has a significantly faster startup time compared to any earlier Sun release.
- Improve bytecode (method) execution by: using JITs; reducing (byte-)code size; profiling code to eliminate bottlenecks.
- Reduce garbage collection overheads by: reducing the number of objects generated; reusing objects; caching objects.
- Reduce multithreading overheads by targeting the granularity of locks, and managing synchronization correctly.
- Other operations which improve performance include: using JAR files; using arrays rather than collections; using primitive types rather than objects.

- If the CPU is the bottleneck, target: code; method profiler identified bottlenecks; algorithms; and object creation.
- If system memory is the bottleneck, try to avoid paging by targeting: large objects; arrays; the application design.
- If disk I/O is the bottleneck, identify the problem and eliminate it.
- Ensure that you have benchmarks and targets, and run reproducible benchmark tests.
- Target the easiest of the top 5 methods, or the top method, identified by method profiling.
- Repeat profile, fix, benchmark iterative process.
- Avoid runtime String concatenation. Use StringBuffer instead.
- Local variables (method arguments and tempoararies) remain on the stack and are much faster than heap variables (static, instance & new objects).
- Use strength reduction: "x = x + 5" -> "x += 5"; "y = x/2" -> "y = x >> 1"; "z = x * 4" -> "z = x << 2".
- Reuse threads by pooling threads.
- Use Buffered I/O classes.
- Method synchronization is slightly faster than block synchronization (and produces smaller bytecode).
- Optimize after profiling the functional application, not before.
- Obfuscators can make class files smaller.

http://www.joot.com/articles/practices.html
Good Java practices, some of which are good for performance. (Page last updated January 2001, Added 2001-01-19, Author Dave Jarvis, Publisher JOOT). Tips:

- Always profile the code to find where the bottlenecks are.
- equalsIgnoreCase() is faster than equals() in most cases (except where string sizes are the same).
- Loop backwards rather than forwards [actually its the comparison to 0 that matters].
- Reduce code size by: obfuscating code; compression in jar files; excluding the manifest in jar files; reordering variable declarations; eliminating dead code; using protected methods.
- Manipulate data that requires parsing into a format that is easier to parse.
- Use bitshift instead of multiplying or dividing by powers of 2.
- Use binary-And (on N-1) instead of modulus (on N).
- Use Thread.sleep() instead of a `for` loop for measured delays.
- Use the update() method to draw things, not the paint() method.
- Use double-buffering.
- Apply faster algorithms and data structures.
- Use StringBuffer for String concatenations, rather than the '+' operator.
- Use static [pre-created] exceptions.
- Use `final` classes.

http://www.sys-con.com/java/article.cfm?id=1169
Javabean component architecture (Page last updated October 2001, Added 2001-10-22, Authors David Hardin and Mike Frerking, Publisher Java Developers Journal). Tips:

- Reusing events reduce object creation and garbage collection overheads.
- Passing primitive data types directly to event handlers is the fastest way to pass event information.
- Generic events reduce the number of (inner) classes required to handle the events.

http://developer.java.sun.com/developer/technicalArticles/Using/
The logging APIs (Page last updated September 2001, Added 2001-10-22, Author Tom Harpin, Publisher Sun). Tips:

- [Article gives a high level view of the logging APIs introduced in SDK 1.4. No application is adequately deployed unless it has some performance logging in place].

http://www.javaworld.com/javaworld/jw-03-2001/jw-0309-games.html
The performance of games on J2ME (Page last updated March 2001, Added 2001-03-21, Author Jason R. Briggs, Publisher JavaWorld). Tips:

- Target performance for processors that you will run on when the project is deployed.
- Implementing the ImageProducer interface and setting an image's pixels directly eliminates one or two steps in the MemoryImageSource option and seems to be about 10 percent to 20 percent faster on average.
- Raw frame rate display, without taking account of the time taken to draw an image, runs from 2 frames per second (fps) to 400 fps, depending on processor and JVM. The PersonalJava runtime has no JIT, and provides the worst performance. With a JIT it might be usable.
- [Article includes references to a number of hardware based Java implementations, i.e. Java enabled CPUs.]

http://developer.java.sun.com/developer/technicalArticles/Threads/applet/index.html
Introductory level article on threading applets (Page last updated March 2001, Added 2001-03-21, Author Monica Pawlan, Publisher Sun). Tips:

- Multi-threaded programs can allow multiple activities to continue without blocking the user.
- Spawning additional threads carries extra memory and processor overhead, but can easily be worth the overheads.
- Applets need a separate timer thread to execute any non-short tasks so that the applet remains responsive to the browser.
- The volatile modifier requests the Java VM to always access the shared copy of the variable so the its most current value is always read. If two or more threads access a member variable, AND one or more threads might change that variable's value, AND ALL of the threads do not use synchronization (methods or blocks) to read and/or write the value, then that member variable must be declared volatile to ensure all threads see the changed value.

http://www.javaworld.com/javatips/jw-javatip18.html
Cute tip on unblocking a blocked socket by sending it data from a timer thread. (Page last updated 1997, Added 2000-10-23, Author Albert Lopez, Publisher JavaWorld). Tips:

- Use a separate timer thread to timeout socket operations
- Instead of killing the blocked socket, send it some data to unblock it.

http://www.sys-con.com/java/article.cfm?id=1135
J2EE design optimizations (Page last updated September 2001, Added 2001-10-22, Author Vijay S. Ramachandran, Publisher Java Developers Journal). Tips:

- For data that changes infrequently (i.e. rarely enough that a user session will not need that data updating during the session lifetime), avoid transactional access by using a cached Data Access Object rather than the transactional EJB (this is called the Fast Lane Reader pattern).

- Don't transfer long lists of data to the user, transfer a page at a time (this is called the Page-by-Page Iterator pattern).
- Instead of making lots of remote requests for data attributes of an object, combine the attributes into another object and send the object to the client. Then the attributes can be queried efficiently locally (this is called the Value Object pattern). Consider caching the value objects where appropriate.

http://www.java-zone.com/free/articles/sf0101/sf0101-1.asp
Choosing a J2EE application server, emphasizing the importance of performance issues (Page last updated February 2001, Added 2001-02-21, Author Steve Franklin, Publisher DevX). Tips:

- Application server performance is affected by: the JDK version; connection pooling availability; JDBC version and optimized driver support; caching support; transactional efficiency; EJB component pooling mechanisms; efficiency of webserver-appserver connection; efficiency of persistence mechanisms.
- Your application server needs to be load tested with scaling, to determine suitability.
- Always validate the performance of the app server on the target hardware with peak expected user numbers.
- Decide on what is acceptable downtime for your application, and ensure the app server can deliver the required robustness. High availability may require: transparent fail-over; clustering; load balancing; efficient connection pooling; caching; duplicated servers; scalable CPU support.

http://www.javaworld.com/javaworld/jw-06-2001/jw-0622-filters.html
Servlet Filters (Page last updated June 2001, Added 2001-07-20, Author Jason Hunter, Publisher JavaWorld). Tips:

- Servlet Filters provide a standardized technique for wrapping servlet calls.
- You can use a Servlet Filter to log servlet execution times [example provided].
- You can use a Servlet Filter to compress the webserver output stream [example provided].

http://www.onjava.com/pub/a/onjava/2001/09/18/jboss.html
Implementing clustering on a J2EE web server (JBoss+Jetty) (Page last updated September 2001, Added 2001-10-22, Author Bill Burke, Publisher OnJava). Tips:

- Clustering includes synchronization, load-balancing, fail-over, and distributed transactions.
- [article discusses implementing clustering in an environment where clustering was not previously present].
- The different EJB commit options affect database traffic and performance. Option 'A' (read-only local caching) has the smallest overhead.
- Hardware load balancers are a simple and fast solution to distributing HTTP requests to clustered servers.

http://developer.java.sun.com/developer/J2METechTips/2001/tt0917.html
Making HTTP connections using background threads. (Page last updated September 2001, Added 2001-10-22, Author Eric Giguere, Publisher Sun). Tips:

- The user interface must always be responsive to the user's interaction.
- The application should respond to input no later than a tenth of a second after it occurs: longer delays are noticed by the user, and make the user interface seem unresponsive. So don't do more than about a tenth of a second's worth of work in the user-service thread in response to any user interface event.

- Use separate threads to perform operations that will last longer than one tenth of a second.
- Provide the user with the option to cancel the operation at any time.
- [Article provides an example of making an HTTP connection following these suggestions].

http://www.macadamian.com/column/column_fastjava.html
Article by Frederic Boulanger with tuning tips. (Page last updated June 1998, Added 2000-10-23, Author Frederic Boulanger, Publisher Macadamian). Tips:

- Choose the best algorithm or data structure.
- Whatever can be calculated outside of a loop should be calculated outside of the loop.
- Try to minimize method calls within a loop.
- Reduce the number of references to an array in loops.
- Store the value of array or array elements in temporary variables and use these in the loop.
- For multidimensional arrays store a reference for the currently accessed row in a variable.
- Store member variables in a local temporary variable in loops.

http://www-4.ibm.com/software/os/warp/performance/javatip.htm
IBM's list of Java performance tuning tips (same page, two URLs). (Page last updated 2000, Added 2000-10-23, Author ?, Publisher IBM). Tips:

- Group native operations to reduce the number of JNI calls.
- Primitive types are faster than classes encapsulating types.
- Avoid excessive writing to the java console.
- Reorder CLASSPATH so that the most used libraries occur first.
- Don't overuse synchronized methods.
- Use int instead of long when possible.
- When possible, declare methods as final.
- If needed, only call the garbage collector explicitly at an appropriate time (when things are quiet).
- Prudent use of zip and jar formats can improve load time.
- Compile java files with the optimizer on.
- Cache frequently used objects when possible.
- Use static final when creating constants.
- Use StringBuffer when doing excessive string manipulations.
- Consider when to use local variables in loops (speed vs. memory?).
- Vectors are more flexible than arrays, but much slower.
- It is faster to add/delete items from the end of the vector.
- Avoid unnecessary casts and instanceof.
- Scope of variables can impact performance.

http://developer.java.sun.com/developer/technicalArticles/ebeans/EJB20CMP/
EJB2.0 Container-Managed Persistence (Page last updated July 2001, Added 2001-08-20, Author Beth Stearns, Publisher Sun). Tips:

- EJB 2.0 Container-Managed Persistence provides local interfaces which can avoid the performance overheads of remote interfaces.

http://developer.irt.org/script/java.htm
FAQ site with a couple of basic performance tips. (Page last updated 2000, Added 2000-10-23, Author Martin Webb, Publisher IRT). Tips:

- FAQ 4002: Use an ImageObserver (method listed in FAQ) to control how and when images are painted during loading.
- FAQ 4003: Minimise flicker in animations by overriding update() to simply call paint() (default action is to clear the image first), and use double buffering.
- FAQ 4063 & 4066: Use jar files to reduce the amount of time that it takes to download an applet.

http://www.javaspecialists.co.za/archive/Issue002.html
Anonymous inner classes (Page last updated December 2000, Added 2002-04-26, Author Heinz M. Kabutz, Publisher Kabutz). Tips:

- Accessing private data members of an outer class, is done using a generated method, which is slower than normal field access. Though HotSpot can inline the access.

http://developer.java.sun.com/developer/qow/archive/153/index.jsp
Minimizing space taken by HTTP downloads (Page last updated October 2001, Added 2001-10-22, Authors Gary Adams and Eric Giguere, Publisher Sun). Tips:

- Use HttpConnection.getLength() to determine the number of bytes needed to to hold the data from a download.
- Use a ByteArrayOutputStream to accumulate results if the content length is indeterminate.
- The best performance is obtained from a 1.1 compliant webserver using persistent connections.

http://www-106.ibm.com/developerworks/java/library/j-jtctips/j-jtc0219a.html
Double-if on multi-CPU (Page last updated February 2002, Added 2002-03-25, Author Phil Vickers, Publisher IBM). Tips:

- Double-if logic fails on multiple CPU machines. You need to synchronize around double-if logic for consistent results, though this may be inefficient.

http://www.javaworld.com/javaworld/jw-05-2001/jw-0504-cache.html
Faster JSP with caching (Page last updated May 2001, Added 2001-05-21, Author Serge Knystautas, Publisher JavaWorld). Tips:

- The (open source) OSCache tag library provides fast in-memory caching.
- Cache pages or page sections for a set length of time, rather than update the page (section) with each request.
- Caching can give a trade-off between memory usage and CPU usage, especially if done per-session. This trade-off must be balanced correctly for optimal performance.

http://www.sys-con.com/xml/article2a.cfm?id=229
Scaling web services (Page last updated June 2001, Added 2001-07-20, Author Simeon Simeonov, Publisher XML Developers Journal). Tips:

- Use bigger, better, faster hardware, but there is a limit to the scalability of a single server: most application performance does not scale linearly with increases in the hardware power.
- Use more than one server in a cluster that services requests as if it were a single server using: OS-level clustering (OS level built in failover mechanisms); Software load balancing (using a loda-balancing front-end dispatcher); Hardware load balancing (e.g. DNS round-robin to different servers).

- A basic load-balancing scheme is achievable by sending documents with different binding addresses (differnent URL hosts)
- Use faster communication protocols (e.g. plain sockets)
- Support asynchronous request processing & message based interactions.

http://developer.java.sun.com/developer/technicalArticles/Threads/swing/
Multithreaded Swing Applications (Page last updated September 2001, Added 2001-10-22, Author Monica Pawlan, Publisher Sun). Tips:

- Use the SwingUtilities.invokeAndWait() and SwingUtilities.invokeLater() methods to put code on the GUI eventr queue.
- Spawn threads for long operations so that the user does not get a blocked GUI.

http://www.microjava.com/articles/techtalk/display?PageNo=1
Timers and low-level GUI display effects (Page last updated September 2001, Added 2001-10-22, Author Roman Bialach, Publisher Micro Java). Tips:

- You need a scheduling mechanism to perform animation, scrolling, updating the display, etc.
- The paint() method on the Canvas is called by the system only if it thinks that it needs to repaint it. So we need another timer to repaint the screen on a regular basis. Use a timer to periodically call repaint().

http://www.javareport.com/html/from_pages/article.asp?id=4702&mon=9&yr=2001
Architecting and Designing Scalable, Multitier Systems (Page last updated August 2001, Added 2001-10-22, Author Michael Minh Nguyen, Publisher Java Report). Tips:

- Separate the UI controller logic from the servlet business logic, and let the controllers be mobile so they can execute on the client if possible.
- Validate data as close to the data entry point as possible, preferably on the client. This reduces the network and server load. Business workflow rules should be on the server (or further back than the front-end).
- You can use invisible applets in a browser to validate data on the client.

http://developer.java.sun.com/developer/community/chat/JavaLive/2001/jl0619.html
Sun community discussion on "Optimizing Entity Beans" with Akara Sucharitakul (Page last updated June 2001, Added 2001-07-20, Author Edward Ort, Publisher Sun). Tips:

- Prepared SQL statements get compiled in the database only once, future invocations do not recompile them. The result of this is a decrease in the database load, and an increase in performance of up to 5x.
- Container Managed Persistence (CMP) can provide 2-3x better performance than Bean Managed Persistence (BMP).

http://www.sys-con.com/java/article.cfm?id=1081
Optimizing dynamic web pages (Page last updated July 2001, Added 2001-07-20, Author Helen Thomas, Publisher Java Developers Journal). Tips:

- Dynamic generation of web pages is more resource intensive than delivering static web pages, and can cause serious performance problems.

- Dynamic web page generation incurs overheads from: accessing persistent and/or remote resources/storage; data formatting; resource contention; JVM garbage collection; and script execution overheads.
- Dynamic content caching tries to mitigate Dynamic web page generation overheads by reusing content that has already been generated to service a request.
- JSP cache tagging solutions allow page and fragment level JSP output to be automatically cached.
- On highly personalized sites page-level caching results in low cache hit rates since each page instance is unique to a user.
- Component-level caching applies more extensively when components are reused in many pages, but requires manual identification of bottleneck components.

http://www.sys-con.com/java/article2a.cfm?id=732
J2ME apps, with a discussion of the needs to balance performance (Page last updated June 2001, Added 2001-07-20, Author Glenn Coates, Publisher Java Developers Journal). Tips:

- J2ME devices have limited processing power, so performance is important and must be considered for the target device.
- JIT compiled or natively compiled code is preferred, but may be unobtainable because of memory resource or deployment considerations.
- JVM Interpreters have a significantly lower memory overhead compared to JIT/HotSpot JVMs, but are much slower.
- Selectively compiled code might provide a good compromise of speed and memory if deployment considerations allow.
- The application does not need to be lightning fast in order to have a responsive user interface. The perception of speed is important, for example, the user interface should give immediate feedback.
- JVM slection for the J2ME device is pivotal to achieving the required performance.
- Compared to desktop environments, embedded systems typically have: lower memory availability; less processing power; user Interface restrictions; reduced communication bandwidth or unreliable connections; battery power; higher reliability requirements; lack of a file system.

http://www.eweek.com/article/0,3658,s=708&a=23125,00.asp
Tuning JVMs for servers. (Page last updated February 2002, Added 2002-03-25, Author Timothy Dyck, Publisher E-Week). Tips:

- Multiple JVMs are often available for a particular platform. Choose the JVM that best suits your needs.
- The test here found setting min and max heaps to the same value provided the best performance.
- Limiting each Sun 1.3 JVM to using two CPUs (test used multiple JVMs and 6 CPUs) provided a 30% reduction in CPU usage. IBM JVMs did not require (or benefit from) this optimization.

http://www.javaworld.com/javaworld/javaqa/2001-12/01-qa-1207-ziprmi.html
Data compression (Page last updated December 2001, Added 2001-12-26, Author Tony Sintes, Publisher JavaWorld). Tips:

- [Article covers how to add zip compression to RMI communications].

http://www.devx.com/premier/mgznarch/javapro/2001/01dec01/sl0112/sl0112-1.asp
Creating Web-based, interactive graphics. (Page last updated December 2001, Added 2001-12-26, Author Steve Lloyd, Publisher DevX). Tips:

- If an applet parameter's [tags in the webpage] length is too long, the Web page's responsiveness begins to bog down. Move all but the essential parameters from the APPLET tag to a dedicated HTTP link between the applet and the servlet. This allows page loading and applet initialization to occur at the same time over separate connections.
- Close java.sql.Statements when finished with.

http://www.javaspecialists.co.za/archive/Issue042.html
Inverting booleans (Page last updated February 2002, Added 2002-03-25, Author Heinz M. Kabutz, Publisher Kabutz). Tips:

- The fastest way to invert a boolean is to XOR it (bool ^= true).
- Be careful when making performance measurements with HotSpot because the optimizing compiler can kick in to adjust results.

http://www.javaworld.com/javaworld/jw-02-2002/jw-0222-designpatterns.html
The Proxy design pattern. (Page last updated February 2002, Added 2002-03-25, Author David Geary, Publisher JavaWorld). Tips:

- Creating images is expensive.
- ImageIcon instances create their images when they are constructed.
- If an application creates many large images at once, it could cause a significant performance hit.
- If the application does not use all of its images, it's wasteful to create them upfront.
- Using a proxy, you can delay image loading until the image is required.
- The Proxy pattern often instantiates its real object, the Decorator pattern (which can also use proxy objects) rarely does.
- The java.lang.reflect package provides three classes to support the Proxy and Decorator patterns: Proxy, Method, and InvocationHandler.

http://forums.itworld.com/webx?14@@.ee6b80a/534!skip=470
Stateful vs Stateless EJBs (Page last updated May 2001, Added 2001-05-21, Author Chuck Caveness, Doug Pardee, Publisher IT World). Tips:

- Stateless session beans can support multiple clients, thus increasing scalability.

http://www.xml.com/pub/a/2002/01/16/deviant.html
Webservices SOAP communications overheads (Page last updated January 2002, Added 2002-02-22, Author Leigh Dodds, Publisher XML). Tips:

- Generating XML produces a large amount of data during communications, but this does not mean that the communication will be the bottleneck.
- Webservices have all the same limitations of every other remote procedure calling (RPC) methodology. Requiring synchronous communications across a WAN is a heavy overhead regardless of the protocol.
- If "Web services" tend to be chatty, with lots of little round trips and a subtle statefulness between individual communications, they will be slow. That's a function of failing to realize that the API call model isn't well-suited to building communicating applications where

caller and callee are separated by a medium (networks!) with variable and unconstrained performance characteristics/latency.
- Asynchronous messaging may be required for efficient webservices.

http://developer.java.sun.com/developer/community/chat/JavaLive/2001/jl0109.html
Sun community chat session on "Threading and Concurrency in the Java Platform" with Thomas Christopher and George Thiruvathukal (Page last updated January 2001, Added 2001-02-21, Author Edward Ort, Publisher Sun). Tips:

- If memory is at a premium, the cost of allocating a Thread object and allocating a stack can be expensive.
- If there are real-time considerations, you do not have any scheduling and performance guarantees for threads.
- Be careful about the number of threads you create: too many and you can exhaust your memory: too few and you don't get the advantages of parallelism.
- Use javax.swing.SwingUtilities.invokeLater() [java.awt.EventQueue.invokeLater()] to schedule work onto the awt thread.

http://www.microjava.com/articles/techtalk/object_lists?content_id=1152
Basic article on a minimal ArrayList implementation, from a micro-Java slant (Page last updated March 2001, Added 2001-04-20, Author Lee Miles, Publisher Micro Java). Tips:

- ArrayLists are the fastest SDK collection class.
- System.arraycopy provides an efficient method for copying arrays.
- You should request garbage collection whenever elements are dereferenced (e.g. the list is cleared).

http://www.sys-con.com/weblogic/article.cfm?id=102
Precompiling JSPs (Page last updated July 2002, Added 2002-07-24, Author Steve Mueller, Scot Weber, Publisher Weblogic Developers Journal). Tips:

- Precompile your JSPs one way or another to avoid the first user having a slow experience.

http://www7b.boulder.ibm.com/dmdd/library/techarticle/0204pooloth/0204pooloth.html
High performance inserts with DB2 and JDBC (Page last updated April 2002, Added 2002-07-24, Author Krishnakumar Pooloth, Publisher IBM). Tips:

- Use SQLJ to get the use of buffered inserts, and modify the code generated from SQLJ to reuse the RTStatement object.

http://developer.java.sun.com/developer/JDCTechTips/2001/tt0518.html#optimizing
Optimizing StringBuffer usage (Page last updated May 2001, Added 2001-05-21, Author Glen McCluskey, Publisher Sun). Tips:

- Pre-size the StringBuffer to the expected result String size where possible.

http://www.javaworld.com/jw-09-2001/jw-0914-access.html
Customized high-speed, fine-grained access control (Page last updated September 2001, Added 2001-10-22, Author Wally Flint, Publisher JavaWorld). Tips:

- [Article discusses an Access control pattern which has no performance penalty].

http://www.allaire.com/Handlers/index.cfm?ID=17266&Method=Full&Cache=Off
Connection Pooling with JRun (Page last updated June 2001, Added 2001-08-20, Author Karl Moss, Publisher Allaire). Tips:

- Establishing an initial connection is one of the most expensive database operations. Use a pool of connections that are ready and waiting for use to minimize the connection overhead.
- Connection pooling is one of the largest performance improvements available for applications which are database intensive.
- Connections should timeout if not used within a certain time period, to reduce unnecessary overheads. Initial and maximum pool sizes provide further mechanisms for fine-tuning the pool.
- JDBC 2.0 supports connection pooling, though a particular driver may or may not use the support. If pooling is supported by the driver, it is probably more efficient than a proprietary pooling mechanism since it can leverage database specific features.

http://www.as400.ibm.com/developer/java/faq/perffaq.html
Some IBM Java performance tips. Although intended for AS/400 Java, many tips are generally applicable (Page last updated ?, Added 2000-10-23, Author ?, Publisher IBM). Tips:

- Minimize the use of synchronized methods.
- Use the -O javac option.
- Minimize object creation, reuse objects.
- Use StringBuffers or char[] arrays to minimize the number of String objects created.
- Use faster accesses. Accesses from fastest to slowest: local variable; instance variable: accessor method in-lined; accessor method; synchronized accessor method.
- Minimize the use of created exceptions.
- Use `static final` when creating constants.
- Use Prepared Statements.
- Store character data in DB2 as Unicode, numeric data as float.

http://www.messageq.com/systems_management/currie_1.html
Monitoring Networked Applications (Page last updated March 2002, Added 2002-04-26, Author Russ Currie, Publisher Message MQ). Tips:

- Use network probes to break down how the network is being used by the various networked applications on it.

http://www.theserverside.com/resources/article.jsp?l=Is-EJB-Appropriate
Deciding whether EJB is appropriate. (Page last updated September 2001, Added 2001-10-22, Author Ed Roman, Publisher The Server Side). Tips:

- An HTTP layer is not always necessary. Connecting directly to EJBs is faster and provides automatic load balancing.

http://java.oreilly.com/news/javaxslt_0801.html
Tips on using XSLT (Page last updated August 2001, Added 2001-10-22, Author Eric M. Burke, Publisher O'Reilly). Tips:

- XSLT transformations are CPU & memory intensive, so cache results wherever possible. Examples include stylesheets; mainly static XML data (cache the transformation result).

http://developer.java.sun.com/developer/Books/J2EETech/ch3.pdf
Rambling discussion of building J.Crew website, in Chapter 3 of "J2EE Technology in Practice" (Page last updated September 2001, Added 2001-10-22, Authors Dao Ren, Dr. Rick Cattell and Jim Inscore, Publisher Sun). Tips:

- Use database connection pooling
- Cache Database Requests
- [Statistics useful for comparison if you are building a business enterprise site: The architecture can handle 8,000 concurrent user sessions; 85 dynamic page views a second; 250,000 unique daily visitors; 8 million hits a day; 1 to 2 second average response time].

http://www.javaworld.com/javaworld/javaqa/2001-11/02-qa-1109-boolean.html
Converting booleans to strings. (Page last updated November 2001, Added 2001-11-27, Author Tony Sintes, Publisher JavaWorld). Tips:

- Use String.valueOf(bool) to convert booleans to strings.

http://developer.java.sun.com/developer/JDCTechTips/2001/tt0925.html
Generating integer random numbers (Page last updated September 2001, Added 2001-10-22, Author John Zukowski, Publisher Sun). Tips:

- [Article explains why ways of generating random integers produces skewed results. Important for correctly simulating a variety of things].

http://www.aikido.ozactivity.com.au/doc/en/administration/server_tune_procs.html
Tuning tips intended for Sun's "Web Server" product, but actually generally applicable. (Page last updated 1999, Added 2000-10-23, Author ? - a Sun document, Publisher Aikido). Tips:

- Use more server threads if multiple connections have high latency.
- Use keep-alive sockets for higher throughput.
- Increase server listen queues for high load or high latency servers.
- Avoid or reduce logging.
- Buffer logging output: use less than one real output per log.
- Avoid reverse DNS lookups.
- Write time stamps rather than formatted date-times.
- Separate paging and application files.
- A high VM heap size may result in paging, but could avoid some garbage collections.
- Occasional very long GCs makes the VM hang for that time, leading to variability in service quality.
- Doing GC fairly often and avoiding paging is more efficient.
- Security checks consume CPU resources. You will get better performance if you can turn security checking off.

http://www.ibm.com/java/education/javahipr/javahipr1.html
Research paper on high performance Java. (Page last updated 1999, Added 2000-10-23, Author Sandeep K. Singhal, Publisher IBM). Tips:

- Use local variables as a first choice for manipulating data.
- Rewrite the loop test so that it uses a comparison to 0.
- Avoid synchronization where possible.
- Reuse and pool objects.
- Avoid throwing exceptions (*not* avoid using try-catch blocks).

- Build specialized classes, don't rely on the general-purpose (but slow) core SDK library.

http://www.javaworld.com/javaworld/javatips/jw-javatip21.html
Use a zip archive to download classes. (Page last updated 1997, Added 2000-10-23, Author John D. Mitchell, Publisher JavaWorld). Tips:

- Use a zip archive to download classes.

http://www.onjava.com/pub/a/onjava/2001/12/12/jms_not.html
JMS & CORBA (Page last updated December 2001, Added 2001-12-26, Author Steve Trythall, Publisher OnJava). Tips:

- Asynchronous messaging is a proven communication model for developing large-scale, distributed enterprise integration solutions. Messaging provides more scalability because senders and receivers of messages are decoupled and are no longer required to execute in lockstep.

http://www.javaworld.com/javaworld/jw-01-2002/jw-0125-overpower.html
Wrapping PreparedStatement (Page last updated January 2002, Added 2002-02-22, Author Bob Byron and Troy Thompson, Publisher JavaWorld). Tips:

- With Statement, the same SQL statement with different parameters must be recompiled by the database each time. But PreparedStatements can be parametrized, and these do not need to be recompiled by the database for use with different parameters.
- [Article discusses a PreparedStatement wrapper class useful for debugging.]

http://www.javaworld.com/javaworld/javaqa/2001-09/02-qa-0921-double.html
String to double (Page last updated September 2001, Added 2001-10-22, Author Tony Sintes, Publisher JavaWorld). Tips:

- Use `Double.parseDouble()` instead of `Double.valueOf(aString).doublevalue()`.

http://www.owlmountain.com/tutorials/NonBlockingIo.htm
Tutorial on non-blocking socket I/O available from JDK 1.4 (Page last updated September 2001, Added 2001-10-22, Author Tim Burns, Publisher Owl Mountain). Tips:

- [No tips, and a rather haphazard tutorial but beggars can't be choosers].

http://developer.java.sun.com/developer/community/chat/JavaLive/2000/jl1212.html
Sun community chat session with Bill Shannon, Kevin Osborn, and Jim Glennon on JavaMail (Page last updated December 2000, Added 2001-01-19, Author Edward Ort, Publisher Sun). Tips:

- You might see a performance increase by using multiple connections to your mail server. You would need to get multiple Transport objects and call connect and sendMessage on each of them, using multiple threads (one per connection) in your application.
- JavaMail 1.2 includes the ability to set timeouts for the initial connection attempt to the server.
- JavaMail tries to allow you to make good and efficient use of the IMAP protocol. Fetch profiles are one technique to allow you to get batches of information from the server all at once, instead of single pieces on demand. Used properly, this can make quite a difference in your performance.

http://www-106.ibm.com/developerworks/java/library/j-diag8.html
Optimizing recursive methods (Page last updated June 2001, Added 2001-06-18, Author Eric E. Allen, Publisher IBM). Tips:

- Try to convert recursive methods into tail-recursive methods.
- You can test if a particular JIT is able to convert tail-recursive into loops with a dummy tail-recursive method which never terminates. If the JVM crashes because of stack overflow, no conversion is done (if the conversion is managed, the JVM loops and never terminates).
- The HotSpot JVM with the 1.3 release does not convert tail-recursive methods into loops. The IBM JVM with the 1.3 release does.

http://softwaredev.earthweb.com/java/article/0,,12082_778571,00.html
Java collections (Page last updated June 2001, Added 2001-06-18, Author Richard G. Baldwin, Publisher EarthWeb). Tips:

- Choose the right structure for the right job.
- ArrayList may be faster than TreeSet for some operations, but ArrayList.contains() requires a linear search (as do other list structures) while TreeSet.contains() is a simple hashed lookup, so the latter is much faster.

http://www.sys-con.com/java/article.cfm?id=1165
The facade pattern for internationalization (Page last updated October 2001, Added 2001-10-22, Author David Gallardo, Publisher Java Developers Journal). Tips:

- If multiple strings will be compared using internationalized comparison, use (and reuse) CollationKeys to manage the comparisons during sorting.

http://www.sys-con.com/java/article.cfm?id=723
Computational planning and scheduling problem solving (not performance tuning) (Page last updated June 2001, Added 2001-06-18, Author Irvin Lustig, Publisher Java Developers Journal). Tips:

- [Article introduces the solving of planning and scheduling problems in Java]

http://m5.peakin.com/html/body_performance.html
Various tips. For web servers? (Page last updated 2000, Added 2000-10-23, Author ?, Publisher ?). Tips:

- Test multiple VMs.
- Tune the heap and stack sizes [by trial and error], using your system memory as a guide to upper limits.
- Keep the system file cache large. [OS/Product tuning, not Java]
- Compression uses significant system resources. Don't use it on a server unless necessary.
- Monitor thread utilization. Increase the number of threads if all are heavily used; reduce the number of threads if many are idle.
- Empirically test for the optimal number of database connections.

http://javazoid.com/OptimizingSqlView.html
Optimizing padded string display (Page last updated June 2002, Added 2002-07-24, Author Gervase Gallant, Publisher JavaZoid). Tips:

- Avoid copying individual string characters. Use the same underlying char array, by using methods like String.substring().

http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html
Runtime.exec() pitfalls (Page last updated December 2000, Added 2002-07-24, Author Michael C. Daconta, Publisher JavaWorld). Tips:

- Runtime.waitFor blocks until the spawned process terminates.
- Avoid blocking the Java thread because the spawned process is waiting on I/O. Make sure you read and write the spawned process's I/O as required.

http://sharkysoft.com/software/lava/docs/lavadocs/lava/clib/stdio/doc-files/optimize.htm
A worked example of optimizing Lava Rocks Java Printf. (Page last updated 1999, Added 2000-10-23, Author ?, Publisher SharkySoft). Tips:

- When adding multiple items to a collection, add them all in one call if possible.
- Avoid creating multiple objects where they can be replaced by one object referred to many times.
- Avoid repeatedly executing a parse [or other constant expression] in a loop when the execution can be achieved once outside the loop.
- Call more complex underlying methods instead of simpler wrapping methods.

http://www.javaworld.com/javaworld/jw-03-2002/jw-0315-jms.html
JMS redelivery (Page last updated March 2002, Added 2002-03-25, Author Prakash Malani, Publisher JavaWorld). Tips:

- Both auto mode (Session.AUTO_ACKNOWLEDGE) and duplicate delivery mode (Session.DUPS_OK_ACKNOWLEDGE) guarantee delivery of messages, but duplicate okay mode can have a higher throughput, at the cost of the occasionally duplicated message.
- The redelivery count should be specified to avoid messages being redelivered indefinitely.

http://www.ibiblio.org/javafaq/slides/sd2000east/javaio/Java_I_O.html
Java I/O tutorial by Elliotte Rusty Harold (author of O'Reilly's Java I/O book). (Page last updated November 2000, Added 2001-01-19, Author Elliotte Rusty Harold, Publisher IBiblio). Tips:

- Sometimes output streams are buffered by the operating system for performance. The flush() method forces the data to be written whether or not the buffer is full. This is not the same as the buffering performed by a BufferedOutputStream. That buffering is handled by the Java runtime. This buffering is at the native OS level. However, a call to flush() should empty both buffers
- It's more efficient to read multiple bytes at a time, i.e use read(byte[]) rather than read().
- The best size for the buffer is highly platform dependent and generally related to the block size of the disk, at least for file streams. Less than 512 bytes is probably too little and more than 4096 bytes is probably too much. Ideally you want an integral multiple of the block size of the disk. However, you should use smaller buffer sizes for unreliable network connections.

http://docs.iplanet.com/docs/manuals/fasttrak/41/servlets/1-using.htm#17322
iPlanet Web Server guide to servlets, with a section at the end on "Maximizing Servlet Performance". (Page last updated July 2000, Added 2001-02-21, Author ?, Publisher Sun). Tips:

- Try to optimize the servlet loading mechanism, e.g. by listing the servlet first in loading configurations.
- Tune the heap size.
- Keep the classpath short.

http://www.ddj.com/articles/1996/9604/9604e/9604e.htm
Paul Tyma's article on low level Java optimizations. (Page last updated 1996, Added 2000-10-23, Author Paul Tyma, Publisher Dr. Dobb's). Tips:

- Speed up the most-used code, such as highly iterated loops and popularly called methods.
- `int` is faster than `long`
- Performance of code that is heavily dependent upon system API calls is largely out of your hands.
- Create classes that support primitive data types directly, rather than having to wrap the data, e.g. a Stack class that directly stores `int`s.
- Inline code.
- Declare methods as static or private to allow inlining.
- Apply code motion (eliminating redundant calculations).

http://www.javaworld.com/javaworld/jw-06-1997/jw-06-plugins.html
Improving applet download time by installing the applet on the client. (Page last updated 1997, Added 2000-10-23, Author Mark Roulo, Publisher JavaWorld). Tips:

- Store your applet on the client machine so that applet download time is absolutely minimal. This is not worth doing for really small applets.

http://developer.java.sun.com/developer/community/chat/JavaLive/2001/jl1113.html
Sun community chat on iPlanet (Page last updated November 2001, Added 2001-12-26, Author Edward Ort, Publisher Sun). Tips:

- Optimal result caching (caching pages which have been generated) needs tuning, especially the timeout setting. Make sure the timeout is not too short.

http://www.ddj.com/articles/2001/0109/0109a/0109a.htm
Developing Scalable Distributed Applications (Page last updated August 2001, Added 2001-10-22, Author Mario A. Torres, Publisher Dr. Dobb's). Tips:

- Use interfaces. Alot.

http://www.javaworld.com/javaworld/javaqa/2001-08/01-qa-0817-static.html
Inner classes (Page last updated August 2001, Added 2001-10-22, Author Tony Sintes, Publisher JavaWorld). Tips:

- Nonstatic member classes must maintain a reference to the enclosing instance, which adds overhead, so use static inner classes where no acces is need to the enclosing instance.

http://www.javaworld.com/javaworld/jw-03-2000/jw-03-javaperf_2.html
Basic performance tuning intro (Page last updated March 2000, Added 2001-03-21, Author Reggie Hutcherson, Publisher JavaWorld). Tips:

- Use a JIT-enabled JVM or HotSpot.

http://www.javaworld.com/javaworld/jw-12-2001/jw-1214-jylog.html
JyLog logger (Page last updated December 2001, Added 2001-12-26, Author Sanjay Dahiya, Publisher JavaWorld). Tips:

- Using JyLog (which uses the JPDA) slows down the JVM executuion time: use standard logging, not JyLog, for deployed applications.

http://developer.java.sun.com/developer/J2METechTips/2001/tt0416.html
Using Timers (java.util.Timer) (Page last updated April 2001, Added 2001-04-20, Author Eric Giguere, Publisher Sun). Tips:

- Timers provide a simple mechanism for repeatedly executing a task at a set interval [with simplicity being the keyword here. Don't look for anything sophisticated like thread interrupt control].

http://www.sunworld.com/sunworldonline/swol-11-1998/swol-11-itarchitect.html
Article on high availability architecture. If the system isn't up when you need it, its not performing. (Page last updated November 1998, Added 2000-10-23, Author Sam Wong, Publisher Sun). Tips:

- Eliminate all potential single-points-of-failure, basically with redundancy and automatic fail-over.
- Consider using the redundant components to improve performance, with a component failure causing decreased performance rather system failure.

http://codeguru.earthweb.com/java/articles/511.shtml
http://soso.mtlab.hit.edu.cn/codeguru/511.shtml.htm
Another "use StringBuffer instead of '+' tip. (Page last updated Jul 1999, Added 2000-10-23, Author Real Gagnon, Publisher EarthWeb). Tips:

- use StringBuffer instead of '+'.

http://www.spec.org/osg/jbb2000/docs/userguide.html
Tuning the SPECjbb2000 Java specmark. (Page last updated 2000, Added 2000-10-23, Author ?, Publisher SPEC). Tips:

- Use Java profilers (-prof, -Xrunhprof) to determine the routines most heavily used.
- Having extra stuff in CLASSPATH can degrade performance on some JVMs.

http://discuss.develop.com/archives/wa.exe?A2=ind0010A&L=DOTNET&P=R28572
Microsoft discussion about csharp garbage collection (the Java clone unsurprisingly has similar issues) (Page last updated October 2001, Added 2001-10-22, Author Brian Harry, Publisher Harry). Tips:

- [No performance tips here. But a fascinating discussion about all the thought that has gone in to csharp GC, only to result in a what already exists in Java].

http://www.sun.com/software/white-papers/wp-optimize/optimize-part1.pdf
http://www.sun.com/software/white-papers/wp-optimize/optimize-part2.pdf
Sun system (not Java) profiling with Sun WorkShop. (Page last updated 1998, Added 2000-10-23, Author ?, Publisher Sun). Tips:

- Select the right combination of compiler options, optimized libraries, and coding techniques.

- Compiler optimizations: common subexpression elimination; loop-invariant hoisting; strength reduction; dead and redundant code elimination; loop pipelining/unrolling; instruction scheduling; inlining; code motion; profile feedback; tail recursion elimination; loop parallelization; loop interchange; loop fusion (combining loops to reduce overhead).
- Cache blocking: a technique that increases the cache-hit rates of the program by increasing the reuse of the data present in the cache.

(Page last updated , Added 2002-10-30, Author , Publisher). Tips:

- x

Additional Sources:

https://docs.oracle.com/middleware/1212/jdev/OJDUG/optimizing_java_projects.htm#OJDUG1937