A New Computing Paradigm

Topics in This Chapter

- The history of Jini
- Sun’s licensing model for Jini
- Getting and installing the software
- Running Jini services from the command line
Chapter 1

The computer on your desk is a remarkable device. It provides computational power, storage, and speed that would have been unthinkable only a few years ago. The fact that this statement is now such a cliché is only a testament to the incredible changes that have occurred in our industry.

Yet despite such advances, the basic structure of our computers remains much the same as it was in the 1950s: We have central processing units, memory, and disks. And, despite the fact that we now use computers with power that would have been unimaginable 40 years ago to play games and balance our checkbooks, we fundamentally interact with these machines in the same ways that our predecessors did: We install software on them, run applications, and manage the (always scarce) disk resources of our systems. A mainframe systems administrator from 1950 would understand these tasks immediately. In fact, in some sense, we’ve all become systems administrators—now each of us has to manage our own machines, doing tasks that would be familiar to systems administrators of an earlier era. The increase in speed and decrease in size has not brought about a qualitative change in how we manage, install, and use these machines.

The situation in desktop computers stands in stark contrast to the situation in telephony. The telephone network has grown exponentially since its humble beginnings, both in terms of raw capacity and the number of phones connected to the network. One can pick up a standard telephone virtually anywhere in the world and, nearly instantly, create a pipeline suitable for voice or data transfer to anyplace else in the world. And while this system’s complexity has grown expo-
nentially, it is largely hidden from its users. My perception of the phone system
doesn’t grow more complex as users in China come on line. The interface for
connecting to any other place in the world stays largely the same, even as more of
the world becomes accessible. And installation and maintenance of the small part
of the global phone network that’s housed in my residence couldn’t be simpler—
I simply plug in new devices and they become reachable from anywhere in the
world. My own minute chunk of the phone network is owned by me and, if such
a term even makes sense here, administered by me as well. And yet everything
still “just works.”

If we expect our computers to be useful—or even useable—as the number of
interconnected devices on the network explodes, we have to reach this level of
reliability, ease of deployment, and ease of administration. The most administra-
tion we have to do with the telephone system is the occasional replacement of a
handset; why then should we take as a given that our current computers and com-
puter networks require so much hand-holding to work properly?

Jini History

In some ways, the vision of Jini is the vision of Java—Jini is really the fulfill-
ment of the original Java idea of groups of consumer-oriented electronic
devices interchanging data and bits of code. Java started out as a language
called Oak, designed as a portable way to write programs for embedded proces-
sors, at Sun Microsystems Labs in 1990. As the project matured, the language
found its way into new types of devices. One of the first experiments resulted in
Oak running on a handheld computer called the Star-7. In this guise, the lan-
guage was used to build interfaces for digital TV and entertainment applica-
tions. In fact, the creator of Oak, James Gosling, decided to write the new
language only after the team had tried to work with C++ and found it too com-
plicated and “programmer-unfriendly.”

In its second incarnation, Oak found its way onto the Web. In 1994, two engi-
neers at Sun, Patrick Naughton and Jonathan Payne, wrote a Web browser com-
pletely in Oak. This browser, called WebRunner, later became the basis for the
HotJava browser, and became famous for its ability to download executable pro-
grams, called applets, from Web servers and execute them securely within the
browser. The language—rechristened Java for its launch in April 1995—was
released with the browser and made Internet history.

Many of the goals of the original consumer electronics vision of Java—the abil-
ity to move code from device to device regardless of CPU type, security, compact-
ness, and so on—made the language a natural for its new home on the Web.
The Jini Vision

Although Java got its start in consumer electronics, it is most commonly thought of as a tool for building applets. The original vision never died within Sun, however. A group of engineers realized that the original idea was still compelling. And although Java makes this idea possible—what with its ability to send portable code from machine to machine and execute it securely—there are other problems that must be solved to make the vision of constellations of easily administered and easily interconnected devices work in practice.

This vision requires mechanisms that we don’t typically associate with desktop computers.

- The software infrastructure for these devices must be incredibly robust. Toasters and TV sets simply cannot fail with a message asking “Abort, Retry, Ignore?” The software must not only allow, but encourage, the development of reliable systems.
- The devices have to support true, effortless “plug and play.” They should be the Internet equivalents of the telephone: You plug them in and they just work, without having to configure IP addresses, set up gateways and routers, install (and possibly remove) drivers, and so on. Upgrades of software are an important issue here—if an administrator has to be called in to upgrade the software for all the TVs in a large hotel, chances are the TVs simply won’t get upgraded.
- Software systems for the Internet age must be evolvable. While creating software for stand-alone devices, such as the CPU in a microwave, is challenging enough, the potential problems are multiplied by the fact that networked devices must be able to communicate with any number of peer devices on the Internet. And, perhaps more problematic, the creator of the original device may not even know about the existence of new devices that will appear later. We’d like our software services and devices to be able to use each other without massive reconfiguration of the network.
- Devices like these will form spontaneous communities: Imagine a digital camera that’s brought into proximity to a color printer. We’d like to be able to simply print the snapshots on the printer, without having to explicitly tell either device about the other. In many cases, the effort that would be required to tell the devices about one another would overwhelm any potential benefit to be
Chapter 1  A New Computing Paradigm

gained from using them together—which is one reason we don’t typically reconfigure our current networks at the drop of a hat. In this world, networking would become much more dynamic, and less the fixed, static organization that we have today.

The vision of legions of devices and software services working together, simply and reliably, had been espoused by a number of researchers and computer industry leaders before. Mark Weiser of Xerox PARC called this vision “ubiquitous computing,” a term meant to connote the ready availability and useability of devices connected to the network. Bill Joy, one of the founders of Sun and the original creator of Berkeley UNIX, believed that the future would continue to hold traditional desktop computers, but also smart “appliances” in homes and vehicles.

With these visions in mind, a group at Sun set out to provide the infrastructure that would bring Java full circle—they intended to build the software layer that would sit atop Java to provide the benefits of reliability, maintainability, evolvability, and spontaneity that such a world would require. These developers set out to define a model that would be easy for programmers to understand, and yet would lend itself to this new way of building software—a way of building software that is probably foreign to many, even those used to writing network-aware code.

This project became known as Jini,1 and many of the people who created and championed it are the same people who originally created and championed certain aspects of Java. Bill Joy would become one of the inspirations and chief supporters of the project, much as he was for the original Java work. Jim Waldo inspired much of the early work on Jini and would become the chief architect and designer of the system. Ann Wollrath invented Java’s RMI facility, and continues her work in the context of Jini, where she has been one of the key designers of the system. And Ken Arnold would define Jini’s transaction and storage models. One of the key Jini players who came from outside Sun is Bob Scheifler, who once led the X Consortium and has defined the Jini lookup and discovery protocols.

There are any number of exotic programming models that have attempted to solve the problems of distributed computing. The Jini designers could have selected one of these models—such as temporal logic, weakly-consistent databases, or agents—as the basis of a system to address the challenges that Jini addresses. But these systems typically require you to “start from scratch” when

---

1. The creators of the system explain that Jini was chosen because it is an energetic and easy-to-remember word that begins with “J” and has the same number of letters as “Java.” Some of them will joke that Jini stands for “Jini Is Not Initials.” The word is pronounced just like “genie.”
learning them, because their programming models are so radical. Fortunately, the designers built their system on a set of core concepts that, at least individually, will be commonplace to Java developers: mobile code, strongly typed interfaces, and separation of interface and implementation. To this mix of old Java standbys they add some new concepts that are unique to Jini. These include a distributed storage model that can be used as a general-purpose facility for storing and retrieving objects. This storage model is based on the Linda system from David Gelernter at Yale. Jini also exposes a concept that has been used “under the covers” in RMI for years—the notion of “leasing” as a means to regulate access to resources on a network. Finally, Jini makes heavy use of periodic multicast to notify cooperating Jini applications and services of one another.

Again, the basic notions in Jini are familiar to any Java programmer. Jini adds a number of incremental changes to Java to extend it to this new world of lightweight distributed computing, but doesn’t require developers to invest in learning entirely new programming paradigms from the ground up.

**Not Just for Consumer Electronics**

Interestingly, many of the attributes that are desirable in consumer devices turn out to be desirable in desktop and enterprise software as well. If you are a network or systems administrator, you’re probably well aware of the problems of maintaining even stand-alone computers, much less a whole network of machines! PC networks have many of the same needs as consumer devices. We’d like them to be truly “plug and play” (not the weak sort of plug and play found on today’s PCs, but real, reliable networking by just plugging into the wall). We’d like them to be evolvable in a consistent and reliable way—if I install an OS upgrade on one machine on the network, I’d like it (and the rest of the machines on the network) to keep working. These abilities are as important in home networks as they are in workgroup networks.

Enterprise systems have even more stringent requirements. Servers are meant to stay running for months or even years, so they need to be reliable. We’d like to be able to upgrade the software services on them without requiring a reboot or otherwise breaking the world. And we’d like to have much more flexibility in how we configure the services on the network—if my database server is underpowered, I’d like to be able to just move it to a new machine without having to touch every client to tell it where the database now lives.

This need for “administration-free” networking goes beyond simple convenience—it can actually allow us to do things that were so difficult as to be impractical before. For example, Jini enables a world in which you can visit a remote site—say, a customer’s place of business—and be able to use all of the
software services and devices on that remote network. Printers, fileservers, and scanners, as well as any number of purely software services, become accessible without any configuration, administration, or driver installation. Jini’s administration-free properties lend themselves to the creation of spontaneous, ad hoc networks of computers, formed whenever and wherever needed.

**Jini Becomes Public**

The Jini project went on at Sun, hidden from public eyes, until *New York Times* technology reporter John Markoff broke the story in a front page article in 1998. Shortly thereafter, Jini—although still officially unannounced—appeared on the cover of *Wired* magazine. The technology was finally introduced to the public on January 25, 1999, with a host of licensees already on board.

These partners are building Jini-enabled services and devices, including disk drives, digital cameras, printers, and scanners. For its part, Sun is rapidly aligning behind Jini, in much the same way it aligned around Java back in 1995.

**Licensees**

At the time of this writing, Jini has already been licensed by nearly 40 companies that span a huge gamut from device vendors to enterprise software companies. These include disk drive manufacturers such as Quantum and Seagate, cellular phone manufacturers Nokia and Ericsson, printer vendors including Xerox, Canon, Epson, and Hewlett-Packard, camera manufacturer Kodak, networking vendors Cisco and 3Com, software producers BEA Systems, Novell, and Inprise, and a huge number of consumer electronics companies including Sony, Sharp, Philips, and Toshiba. Jini has also been licensed for use by companies as diverse as AOL and Kinko’s.

**Community Source Licensing**

Sun provides access to Jini source code under what is called the Sun Community Source License (SCSL, often pronounced “scuzzle”). This license is designed to provide virtually free and easy access to all the Jini source code by developers, while allowing Sun to ensure that Jini products remain compatible with one another, and that the Jini source doesn’t “splinter” into incompatible versions.

The community licensing model has many similarities to the “Open Source” movement. The Open Source movement espouses freely-available source code, available to anyone who needs it. This approach to software development has
many benefits—look at the rapid development of large software systems such as Linux, for example. But it can have drawbacks. Most importantly, most Open Source arrangements either have minimal controls over the definition of the software (meaning that anyone can introduce an incompatible change), or they require anyone using the software to submit modifications they make back to an organizing body. The first drawback means that the original developer of the technology may be unable to prevent it from splintering into a morass of incompatible versions. The second drawback means that many companies who might otherwise be eager to develop for the technology may be reluctant to be required to turn over their intellectual property to a third party.

The community license model allows very open access to source code, while promoting compatibility among the developer community. And yet it does not require companies or developers to hand their developments back to Sun. How does it work? The SCSL defines two levels of usage of the source code; your responsibility to ensure compatibility increases as you move to the “higher” license level.

At the first level, you can obtain the source code for Research Use. This allows you complete access to the source for personal, research, and educational uses, or to evaluate the source for possible deployment. You can also distribute examples to other research use licensees. The license itself is just a “click through” page on the Sun download site; in fact, the Jini downloads that you get from Sun all include source code. At this level, there are practically no responsibilities imposed on you; primarily, Sun asks only that you notify it if you find bugs in Jini. You are free to learn about the technology, modify the source, create new implementations of the Jini components, and try it out in your applications. You can even create “clones” of the code, as long as you include the proper notices in the source.

When you get to the point of deploying your Jini project, whether internally or as a commercial product, then you move up to the Commercial Use license. This license grants you the right to distribute modified versions of Jini, as long as they conform to certain compatibility requirements. You should note that you can move to this level of access and responsibility without any extra legal overhead—there is nothing to sign, and you do not need to notify Sun of your new use of the technology. The only added responsibilities are that your modified code conform to the Jini naming conventions, that you openly publish the specifications to any extensions you may develop, and that your new implementation pass the Jini Platform Compatibility Kit. These requirements are intended to ensure that you will have complete interoperability and compatibility with any Jini services and applications that may be used with your code.
Chapter 1  A New Computing Paradigm

If you are a SCSL license, your Jini-based product can carry the “Jini compatible” logo if it passes the tests in the Jini Compatibility Kit and you sign and return to Sun a copy of the Jini Logo License agreement. There is no fee to use this logo (under the terms of the earlier Jini license, used in versions prior to 1.1, Sun charged a minimal fee for use of the logo—this charge has been done away with for version 1.1).

These guidelines allow you to have unfettered access to the source code for learning, experimenting, and teaching. As your code becomes more and more public, though, the community source license requires that your responsibilities to maintain compatibility increase as well. There are no requirements for you to hand over any of your code to other developers or to Sun. If you wish, however, you can choose to donate any code you write back to the community, in which case, Sun or other developers can redistribute it.

What does this mean for you? If you’re developing a service for Jini that only depends on the Jini run-time framework and the Jini binaries as posted on the Sun Web site, you can distribute your program without having any source license at all, although you cannot redistribute the Jini binaries themselves (Sun requires a commercial license to redistribute their binaries). The SCSL model deals with access, modifications, and redistribution of the Jini source code, not the binaries. Only if you plan to use the source in your applications or make modifications to it will you need to follow the guidelines in the SCSL.

Sun intends the community licensing approach to afford the best of both worlds—easy and relatively unfettered access to source code, while still ensuring some degree of compatibility and co-evolution of the code. The SCSL process is still evolving, and Sun is actively soliciting feedback on their licensing strategies. The license has grown considerably simpler and less restrictive, just since the Jini 1.0 days.

Since this is my interpretation of the license, and I’m not a lawyer (thankfully!) you should read up on the full details of Jini’s community licensing model at Sun’s Web site. If you have any questions about the SCSL, you should consult the FAQs and other information at this page.

http://www.sun.com/jini/licensing/

Getting and Installing Jini

Obviously before you can start programming with Jini, you need to get the software set up! Chances are, if you’re a Java developer, you’ve already got Java 2 installed on your system. If you don’t have this version (or a later Java 2
Getting and Installing Jini

point release), you’re going to need it, because Jini requires some features that are only in Java 2.

Here’s the checklist for getting Jini up and running:

1. Install Java 2, if you don’t have it already.
   • Download the Java 2 software from Sun
   • Unpack the distribution
   • Examine the distribution
2. Install the Jini release.
   • Download Jini from Sun
   • Unpack the distribution
   • Examine the distribution
3. Set up your environment.
   • Make sure the Java binaries are in your PATH
4. Start the Jini run-time services, using either the GUI or the command line.
   • Configure and start the HTTP server
   • Configure and start the RMI activation daemon
   • Configure and start the Jini Lookup Service
5. Run the sample programs to make sure you’ve got everything correctly configured.

The remainder of this chapter details the steps required to download and install the Java 2 JDK (Java Development Kit) and Jini. The instructions for setting up Java 2 are particular to the Windows and Solaris implementations of the JDK that are available from Sun; if you run on a different platform—or simply use a different JDK implementation on one of these two platforms—then you should check with your operating system vendor for instructions on how to download and install Java 2.

Since the sample implementation of Jini that comes from Sun is itself implemented completely in Java, there is one version of the code for all platforms. The installation instructions given here for Jini should be useful for all machines and operating systems.

Core Note: Jini sample implementations and specifications

Jini, much like Java, is defined as a set of core specifications that detail how the software must work. And just as any operating system vendor or group of developers can create an alternate implementation of Java that conforms to these specifications, there
can be multiple implementations of Jini as well.

But just as Sun provides “sample” implementations of the Java technology for Windows and Solaris, it also provides a sample implementation of the Jini specifications. Sun’s implementation is pure Java, and so should run on any platform with a fully conformant Java 2 implementation. This Jini sample implementation has been tested on the reference implementations of Java for Windows and Solaris; the Jini development team actively encourages developers to test Jini on other JDK implementations.

The installation and set-up instructions here detail the procedures for the sample implementations of both Java and Jini.

---

**Installing Java 2**

First, let’s look at downloading and installing Java 2.

**Downloading the JDK from Sun**

The most current Java 2 JDK releases are always available from:

http://java.sun.com/jdk/

Click on the appropriate link for your operating system (currently either Windows or Solaris) and save the resulting file to your hard disk. There is also a documentation bundle separately available from the download site. You may wish to copy this bundle—which is available either as a ZIP file or as a compressed “tar” file—to your local system, if you don’t otherwise have access to the Java 2 documentation.

**Unpacking the Distribution**

Next you need to unpack and actually install the JDK. In all of the examples in this book, I will use Java 2 version 1.3 (also called just JDK1.3). The examples should work fine with the earlier 1.2 release also though.

For Windows: If you’re on the Windows platform, you’ve just downloaded a self-installing EXE file. You can just double-click on the file to install it. The installation script will copy two separate sets of files to your hard disk: the first is the JDK proper, which contains the compiler, debugger, and associated tools. The second is the Java Runtime Environment, which contains the Java Virtual Machine (JVM) and the class libraries for the language. In almost every case, simply installing these files to the default location suggested by the installer is fine.
Getting and Installing Jini

For Solaris: Solaris users wind up with a self-installing shell script. You can run this script by invoking `sh` (the UNIX command shell interpreter) on the file you downloaded. For example, if you named the file `Solaris-install.bin`, you would type:

```
sh Solaris-install.bin
```

You’ll be asked to accept the license agreement and provide a directory to unpack into.

Examining the Distribution

Once you’ve downloaded and installed the JDK, you should probably familiarize yourself with where all the important pieces live. Table 1–1 presents a list of the most important components of the JDK; I’ll assume that you unpacked the contents into a directory called `jdk1.3`. It’s important to understand where the essential components live, so that you can make sure your environment is properly configured for running Java.

<table>
<thead>
<tr>
<th>Table 1–1 Layout for the JDK 1.3 Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdk1.3</td>
</tr>
<tr>
<td>jdk1.3/bin</td>
</tr>
<tr>
<td>jdk1.3/demo</td>
</tr>
<tr>
<td>jdk1.3/doc</td>
</tr>
<tr>
<td>jdk1.3/jre</td>
</tr>
<tr>
<td>jdk1.3/jre/ext</td>
</tr>
<tr>
<td>jdk1.3/lib</td>
</tr>
</tbody>
</table>
Chapter 1  A New Computing Paradigm

**Installing Jini**

Next, let’s look at installing the Jini sample implementation on your system.

**Downloading Jini from Sun**

As of the writing of this book, the Jini code is available from the Java Developer’s Connection Web site. This site requires a free registration before you can access the code there. (In general, if you’re a Java developer, you should probably register at this site—it’s the source of early access Java releases from Sun, bug listings, and many helpful articles on Java development.)

The version of Jini that this book is written against is called the “Jini System Software 1.1” release, or just “Jini 1.1.” You may want to read the section, “A Note on Versions” at the end of the Introduction for a quick overview of what Jini versions are available and what has changed with each version. Before downloading from the Java Developer’s Connection site, you should also check Sun’s home page for Jini software to see if newer releases have become available since the writing of this book. The Jini home page is at:

http://www.javasoft.com/products/jini/

To download the software from the Developer’s Connection, register with the site (if you haven’t already), and then go to:


This page has information on the latest versions of Jini, as well as information on how to submit bugs or requests for improvements. The “Product Offerings” link is where the actual downloadable Jini code comes from. There are a number of separate Jini-related downloads available, including:

- Jini Starter Kit. This download has the fundamental pieces of the Jini infrastructure, including the interfaces and library code that allow Java programs to interact with the key Jini services, as well as the Jini specifications for those services. This package includes not only public interfaces to the Jini software, but also implementations of several key Jini services, including—as of release 1.1—the JavaSpaces “storage service,” which I’ll discuss in Chapter 16. The Starter Kit also includes source code for the basic Jini classes.
• Jini Technology Compatibility Kit. This download contains code that can be used to test the compatibility of Jini services and applications. If you create custom services, or reimplement any of the core Jini interfaces, you should download the compatibility kit to test that your code will work properly when deployed against the Jini reference implementation.

At this point, you only need to download the Jini Starter Kit. In general, the compatibility kit is most useful for developers writing commercial Jini services or creating custom reimplementations of the core Jini functionality.

Unpacking the Distribution

Once you’ve downloaded the Jini Starter Kit ZIP file, which is called jini1_1.zip, you’re set to unpack and install it. If you have decided to download the compatibility kit at this time, you should first install the basic Jini Starter Kit, and then install the other package “on top” of it. Again, however, you only need the Starter Kit, so here I’ll only talk about installation of this package.

For Windows: You can use your favorite ZIP extraction utility, such as WinZip. If you want the Jini code to live under the C:\ directory on your PC, you can extract to C: and the unzip utility will create a directory called jini1_1 to hold the extracted software. Open the Starter Kit ZIP file and extract the contents to your desired directory.

For example:

```bash
cd c:\
unzip -d jini1_1.zip
```

For Solaris: Copy the ZIP files for the Starter Kit to the directory you want to contain the software, for example, /files, then unzip the file.

```bash
mkdir /files
cd /files
unzip jini1_1.zip
```

(Note that the -d option is not needed on Solaris.)

This process will create a directory called jini1_1 under the directory in which you ran unzip.
Exchanging the Distribution

You'll use the code here pretty extensively during the course of this book, so you should make sure you understand what you've just unpacked and where everything lives. Table 1–2 shows the most important contents of the jini1_1 directory.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jini1_1</td>
<td>This is the root of the Jini installation.</td>
</tr>
<tr>
<td>jini1_1/index.html</td>
<td>The index.html file is the root of all the documentation that ships with the Jini release.</td>
</tr>
<tr>
<td>jini1_1/doc</td>
<td>This directory contains documentation for the Jini release, including JavaDocs for the APIs, specifications, a glossary, and hints on running the Jini examples.</td>
</tr>
<tr>
<td>jini1_1/doc/api</td>
<td>This directory contains the API documentation for the system.</td>
</tr>
<tr>
<td>jini1_1/doc/specs</td>
<td>All the Jini specifications, which describe the Jini distributed computing model and all the core Jini services, live here.</td>
</tr>
<tr>
<td>jini1_1/example</td>
<td>The 1.1 release of Jini ships with a number of example programs. These include a ray tracing example and a distributed book bidding application. While the actual code for these examples lives in the lib directory, along with the rest of the Jini implementation, this directory contains security “policy” files that support these applications.</td>
</tr>
<tr>
<td>jini1_1/lib</td>
<td>This directory contains the JAR files that constitute the Jini implementation and interfaces. The jini-core.jar, jini-ext.jar, and sun-util.jar files contain the basic Jini interfaces that application writers will use. Other JAR files include the examples, the implementations of the core services, and some utility classes.</td>
</tr>
<tr>
<td>jini1_1/source</td>
<td>This directory contains the source code for the Jini distribution, under Sun’s community source license.</td>
</tr>
</tbody>
</table>
Once you’ve downloaded and unpacked the files, you should read the `index.html` file to make sure that the installation and configuration instructions have not changed—the information below describes how to install and configure Jini for the release that was current at the time this book was written.

**Set Up Your Environment**

In this book, I’ll follow a number of conventions when describing how to compile and run the example programs. One of those conventions is that I never set the CLASSPATH environment variable. Instead, I explicitly pass the JAR files and directories that a program may need on its command line via the `-cp` option. I’ll go into more detail in Chapter 5 on the reasons for this, as well as other aspects of what I call “developing for deployment”—these are some simple rules to follow that can help identify configuration problems early.

The good news at this point though is that there’s very little configuration you need to do to set up your environment to use Jini. In fact, at this point I’d recommend that you *unset* your CLASSPATH and just follow the instructions below to make sure your PATH is set correctly.

**Setting Your PATH**

To run any Java programs, you need to make sure that the `bin` directory from the Java 2 release is in your PATH. Setting your PATH correctly will ensure that all of the java binaries—including the compiler and RMI stubs generator—are accessible; these binaries will “automatically” augment your CLASSPATH to find the basic Java libraries when they are run, so you don’t need to set any special CLASSPATH to use them.

*For Windows:* Type the following:

```
set PATH=c:\jdk1.3\bin;%PATH%
```

In this and later examples, you can change your Windows environment permanently by updating the Environment control pane under the System Control Panel.

*For Solaris:* The particular syntax varies depending on the shell that you use. If you use the common C-Shell, use the following command line and then type `rehash` to cause your change to the PATH variable to be reevaluated:

```
setenv PATH=/usr/java1.3/bin:${PATH}
```

You can make this change permanent by editing your `.cshrc` file.
Chapter 1  A New Computing Paradigm

Core Note: The JAR files in the Jini distribution

If you’re curious, you’ve probably already poked around in the Jini distribution by now. The most important files here are the JAR files in the jini1_1/lib directory, in particular the files jini-core.jar (which contains the core interfaces and classes that make up Jini), jini-ext.jar (“extra” software, which will certainly prove useful to you, but aren’t “annointed” as part of the central, core software package), and sun-util.jar (which contains some useful Sun-provided classes that are subject to change in future releases).

You’ll no doubt notice that there are a number of other JAR files in the lib directory as well; many of these end in “-dl”. These JAR files contain code that is meant to be downloaded to clients as they run.

So, for example, you’ll see reggie.jar (which contains Sun’s implementation of the Jini lookup service) as well as reggie-dl.jar (which contains only the code that clients need to use the lookup service).

When you begin to create your own Jini applications, you may wish to make a similar separation, with the implementation of your application in one JAR file and the parts that will need to be downloaded to clients in another.

Start the Jini Run-Time Services

You’ve probably skinned through the preceding sections with no problems whatsoever. Now, though, you’re going to need to slow down a bit and take the time to start up the services that Jini needs at run time. The bad news is that figuring out the parameters and configurations needed to run these services can be a bit of work. The good news is that you rarely—if ever—have to restart them once they’re going.

Jini depends on quite a bit of network infrastructure at run time. In a deployment setting, you would probably run most of these services on a compute server, or perhaps several compute servers for redundancy. To use Jini effectively, you will need one instance of most of these services running somewhere on your LAN; other services are more effectively deployed on all (or at least several) hosts. If you’re doing your own development and experimentation with Jini, you can just run these services on your desktop computer, if you like.
The Jini distribution comes with the basic services needed to support Jini, as well as a few other “optional” services. The list below enumerates the various services that are required to run Jini applications on your network. In addition to these, there are a few other services—including the Jini “transaction manager” and the JavaSpaces “storage service,” among others—that come with the distribution and that you should be aware of, but aren’t required for most Jini work. So I won’t talk about these until later in this book. But for development, you should go ahead and start all of the required services as described below. You can leave them running indefinitely, and pretty much forget that they’re there.

The services that Jini requires are as follows:

- A simple Web server. Jini requires this facility because when downloaded code is needed through RMI, the actual transmission of the code happens via the HTTP protocol. Jini comes with a very simple HTTP server that’s sufficient to supply code to applications that need it. A common configuration is to run an HTTP server on each host that needs to provide downloadable code to other applications.

- The RMI activation daemon. Despite its frightening name, the activation daemon is a very simple-to-use and useful piece of Java infrastructure. This process allows objects that may be invoked only rarely to essentially “go to sleep” and be automatically awakened when they are needed. This situation commonly arises in remote systems programming, where you may have a long-lived server object that is only rarely used. The RMI activation daemon manages the transition between active and inactive states for these objects, and is used extensively by the other core Jini run-time services. At a minimum, you will need to run the activation daemon on each host that runs a lookup service, described below.

- A lookup service. As you shall see, the lookup service is really the core of Jini. A lookup service keeps track of the currently active Jini services that are available on a LAN. Sun provides its own implementation of a lookup service that’s custom-built for Jini. When you read the Jini documentation, you may notice that the RMI registry server that comes with the JDK can be used as a lookup service. While this option can be used, it is not recommended—the lookup service that comes with Jini is much more full-featured. I won’t examine how to use the RMI registry as a lookup service in this book, because there is no good reason
to use it. The lookup service relies on activation to recover its state after crashes or restarts. So you must run the activation daemon on each machine that runs a lookup service.

This may seem like a lot of work, but these processes can be spread across the machines in your network, and are largely self-maintaining: Once you start them, they need virtually no caretaking. In the course of my Jini development, I leave these processes running for weeks or even months. Many times I’m surprised to see one still running on a server that I’d completely forgotten about.

There are two ways to start these services. If you want to get up and running quickly, you can use a graphical user interface (GUI) that comes with the Jini distribution. This interface is only capable of starting Jini services on the local machine, which is definitely not appropriate for debugging Jini applications that will be run in multimachine environments! Still, the GUI can help you get started more quickly than you could otherwise. I’ll discuss this method first.

Most serious developers, though, will want to get “under the hood” to understand how to run the individual services “by hand” and pass any necessary arguments to them. So we’ll also look at how to do this.

The next section discusses running the required Jini services via the GUI; the following section gives more in-depth guidelines for running these services from a command line on either Windows or Solaris.

### Starting Required Services via the GUI

Jini ships with an easy-to-use GUI for starting the required Jini services. When you use this interface, you start these services on the local machine—typically the same machine on which you will be doing development. The GUI doesn’t hide all the details of these services—it just makes the most common configurations easy.

If you wonder about the various options for these services, read the next section on how to start these services from the command line. In general, having an understanding of the options for these services will be useful as you get more skilled in using Jini—so understanding how to start these services from the command line is important.

---

**Core Note: A new service launcher has been added in Jini 1.1**

If you’ve been using Jini since the 1.0 days, you’re probably familiar with the use (and limitations) of the old StartService GUI application that came with 1.0. One of the most severe problems with this application was that the default parameters for starting different
Getting and Installing Jini

services was hard coded into the application—meaning that you would have to edit and recompile it if you wanted to change the defaults.

In 1.1, Sun has added a new, and greatly improved, service launcher in the package com.sun.jini.example.launcher.StartService. This new launcher lets you load pre-defined service configurations from property files, make changes, and then save these configurations. This new tool isn’t just a Jini service launcher—it can be used to start any Java program you desire.

Although Jini 1.1 also ships with the old service launcher (in com.sun.jini.example.service.StartService), Sun’s intent is to have the new launcher become the “standard” GUI for launching the Jini services. The old launcher will probably disappear in the next release.

Running the GUI

Once you’ve got your PATH set up, you can easily launch the configuration application. To do this, just run the following:

On Windows:

```
java -cp C:\jini1_1\lib\jini-ext.jar;
    C:\jini1_1\lib\jini-examples.jar
    com.sun.jini.example.launcher.StartService
```

On Solaris:

```
java -cp /files/jini1_1/lib/jini-ext.jar:
    /files/jini1_1/lib/jini-examples.jar
    com.sun.jini.example.launcher.StartService
```

Note that although I’ve broken the commands into separate lines for clarity, you should type all of the command on a single line. Be sure, of course, to also use the correct directories for your Jini installation, if you’ve installed Jini into someplace other than I have.

The -cp option sets the CLASSPATH for the JVM to include the jini-examples and jini-ext JAR files; the last argument is the fully qualified name of the StartService GUI.

Once this program launches, you’ll see a window with two tabbed panes, as shown in Figure 1-1. The first is labeled Template and the second is labeled Run.
Chapter 1  A New Computing Paradigm

The Template tab is where you would create launching instructions for new services that you want to use the GUI to start. The Run tab will contain buttons to start and stop any service that has a template defined for it.

![Figure 1-1](image)

**Figure 1-1**  *The StartService launcher*

To begin, go to the File menu and select Open Property File. Jini 1.1 comes with property files that contain pre-defined templates for launching services on both Windows and Unix. The Windows property file can be found as

C:\jini1_1\example\launcher\jini11_win32.properties

The Unix property file lives in:

/files/jini1_1/example/launcher/jini11_unix.properties

Navigate down the directories and load the appropriate property file for your platform. After loading the property file, the GUI will display a separate tabbed pane for each service defined in the property file, and the Run tab will change to
display start and stop buttons for each of the defined services. Figure 1-2 shows how the GUI should look after you’ve loaded the property file.

Figure 1–2 The StartService GUI after loading a property file

Flip through the panes and take a look at the configuration options for these various services. If you’ve installed Jini 1.1 in the “standard” locations, the defaults should work fine for you as-is. If not, you may need to tweak path names in a few places.

For most of the examples in this book, you will need to run only a subset of the services listed here. The most important of these is the Jini lookup service. Additionally, and as I mentioned before, you’ll need a couple of “supporting” services that the lookup service needs to work properly. In particular, the RMI activation daemon (`rmid`) is needed, because the lookup service is implemented using the RMI activation framework. The activation daemon must run on the same machine as the lookup service, and must be started before the lookup service. Additionally, an HTTP server that provides access to the core Jini JAR files is also needed, somewhere on the network.
Chapter 1  A New Computing Paradigm

Take a look at the RMID tab to get the configuration panel for the activation daemon. You can see this panel in Figure 1-3. As you can see, there’s not a lot to be configured! The RMID Command field specifies the command to run, and the Options field sets the security “exec policy” for the activation daemon (I’ll describe this in more detail a bit later in the chapter). The only thing you may want to add is to give rmid a fixed log directory. By default, rmid creates a new log directory below whatever directory is current. It uses this log information to restart any activatable services the next time it is run. To be sure that you’re getting the same log directory each time, you may want to explicitly set it in the Options field of the RMID configuration panel. You would do this by entering -log directory in the panel, where directory is replaced by a fully-qualified path to some existing directory.

![The RMID configuration panel](image)

**Figure 1–3** The RMID configuration panel

Go to the WebServer configuration panel next, as shown in Figure 1-4. The Java Options and JAR File text areas are simply used to specify which JAR file contains the code for the web server. Make sure the path to the JAR file points to the tools.jar file that comes with Jini. The Port and Document Area tabs
allow you to change the default port on which the web server runs, and the
directory from which it will serve documents. You should probably leave these
alone, with the exception of changing the web server port to 8080 (which I’ve
done here with consistency with the rest of the book, and the other installation
instructions that come with Jini). Of course, if you’ve installed Jini in some
unusual location, you’ll need to make the appropriate changes. Finally, the Log
Downloads option passes the -verbose flag to the web server. This will cause
it to log all attempts to download files to standard output. I recommend leaving
this option set to -verbose as it can be a great debugging aid.

![Figure 1–4](image)

**Figure 1–4** The WebServer configuration panel

---

**Core Note: A word on port numbers**

You may have wondered why I recommend changing the port
number for the web server to 8080 in the instructions above. In all
Jini releases prior to 1.1, port 8080 was the “recommended” port to
use. Of course, you can use any port you wish for your webserver, as
long as you’re consistent. But all of the Jini documentation and
installation instructions always used 8080 in the descriptions.
Chapter 1  A New Computing Paradigm

With 1.1, a few inconsistencies crept into the setup instructions. The StartService GUI uses 8081 while the instructions for running the services “by hand” still use 8080.

In the interests of consistency in this book, I’m recommending that you use port 8080 if possible (that is, unless you’ve already got something else running at that port). This ensures that—whether you use the GUI or start things by hand—you’ll be using the same configuration, and the rest of the instructions in the book will “just work.”

Be sure to note that there are a few other places in the GUI you’ll have to make this change.

Next, check out the Reggie configuration panel, as shown in Figure 1-5. This panel lets you configure the Jini lookup service. The Security Policy File option sets up a security policy file for the lookup service. The notion of security policies was introduced in Java 2 as a way to provide fine-grained control over the rights of Java code running in a JVM. If you’re interested in the details of why this is needed for particular services, see the longer descriptions in the section on
Getting and Installing Jini

starting services from the command line. For now, however, all you have to do is edit the path here to point to the location of the policy file that ship with Jini.

The **Executable JAR File** option should indicate the path to the `reggie.jar` file in the Jini distribution. The **Log Directory** specifies a path to a directory in which the lookup service will checkpoint its state. The lookup service can recover from crashes by using the information here. The lookup service will create the child-most directory (called `reggie_log` by default), so it shouldn’t already exist. All the directories leading up to it should exist, however.

The **Groups** field lets you specify which communities this particular lookup service will support. You’ll learn more about what this means in future chapters but for now, just leave it set to `public`.

The one option here that you will have to tweak is the **Codebase** field. This field specifies a URL from which clients of the lookup service can download the code needed to interact with the lookup service. You must provide a URL here that corresponds to the HTTP server you configured in the last step. In particular, you need to replace the string `HOSTNAME` with the name of your computer, and set the port number to 8080 as described in the Core Note previously.

![Figure 1–5  The Reggie configuration panel](chap1.fm)
You will not need to use the panels for the other services now. These are used to control the Jini transaction manager and JavaSpaces services, as well as a handful of Jini utility services; you’ll see how to start these later in the book.

Finally, after all this, you’re set to run! Click on the Run tab, which brings up the window shown in Figure 1-6. You can now click on Start RMID, Start WebServer, and Start Reggie in that order (make sure that you start the reggie lookup service last, as it depends on the web server and activation daemon to be running).

Core Tip: Understanding the lifecycle of the lookup service

Chances are, you’ve never used an activatable program like the Jini lookup service before. What happens when you “start” an activatable service like reggie is that the launching program doesn’t actually start the lookup service at all. Instead, it’s merely a setup program that “registers” it with the activation framework, so that reggie can be started on demand.

The activation daemon, in turn, records information provided by
the setup program so that it can start reggie in the future as needed. This is the information that’s squirreled away in the activation daemon’s log directory.

Why do you need to care about this? Most importantly, this means that you generally need to only run the lookup service once, ever. “Running” the lookup service actually just does this registration process, which is typically a one-time affair. After this, if your machine ever crashes or reboots, simply restarting rmid will cause the lookup service to be launched...as long as rmid can access its log files of course. Re-running the setup program just registers multiple copies of the lookup service on the same machine, which is probably not what you want.

The later section on running the services by hand has more details on reggie lifecycle and interactions with activation.

At this point, you’re set to start running some Jini examples. As mentioned earlier, getting the various services set up can be a pain, largely because of all the options that are needed, and uncommon JVM parameters (such as security policies) that have to be used. But once you’ve set up these services, you can likely leave them running for long stretches of time.

If you do want more details on the service launcher, read the file NewStart-Service.txt in the directory jini1_1/doc/example that comes with the distribution.

For those with a serious interest in how the Jini services work, I’ll now present the details of starting these services from the command line, along with some helpful scripts for getting things going. If you’re going to be starting these services automatically when servers boot, you need to understand how to launch them manually. The next section also looks behind the curtains at some of the options that I glossed over in the GUI walk-through.

**Starting Required Services via the Command Line**

If you have already started the required services from the GUI, you can skip over this section if you like. The purpose of spelling out how to run these by hand is to give you a bit more information about what the options for the various services are, and what common configurations of the Jini services are useful. And knowing how to start these services by hand is essential if you need to configure your machine to start Jini services at boot time.

This is the nitty-gritty on how to run these components. I’ll walk through the steps required to start these services. In general, you should start them in the...
Chapter 1  A New Computing Paradigm

order listed. For each of these services, I’ll provide short scripts for both Windows and UNIX that will save you time.

Start the Web Server

As mentioned, to support the downloading of code, you need to have an HTTP server that can provide the code needed by applications running somewhere on your network. The requirements of this server are minimal—it only really needs to support the “get” operation, so any old Web server that you’re already running is probably sufficient.

Jini comes with a utilitarian Web server in its tools.jar package, though, so I’ll show you how to get it running.

In the most basic case, all you have to do is type the following to run the HTTP server:

On Windows:

```java
java -jar C:\jini1_1\lib\tools.jar
   -port 8080 -dir C:\jini1_1\lib
   -verbose
```

On UNIX:

```java
java -jar /files/jini1_1/lib/tools.jar
   -port 8080 -dir /files/jini1_1/lib
   -verbose
```

Obviously, substitute whatever directory in which you have installed the Jini release. Here you see an example of an executable JAR file that has a default program that will be run when it is launched by the JVM.

There are a number of options you can pass to this server to customize its behavior. By default, the HTTP server runs on port 8080. If you already have a Web server running there, you may wish to launch the Jini HTTP server on a different port by passing the `-port <portnum>` option, as shown here. You can also set the root directory that will be served by passing `-dir <directory>` to the server; here I’ve shown the HTTP root directory to be set to the Jini `lib` directory, so that all of the core Jini code can be exported. Finally, the `-verbose` option is useful for debugging—it will cause the HTTP server to display each request made to it as well as where the request came from. You may wish to check the documentation that comes with the Jini release for more options, or in case you have specific configuration needs.

In general, any code that may need to be downloaded across the network has to be accessible from an HTTP server. This server doesn’t have to be a full-
Getting and Installing Jini

blown, general-purpose Web server; it can be as simple as the HTTP service that comes with Jini. You can collect up all the downloadable code together and have one Web server serve it, or you can have a number of small servers running on your network. For development purposes, since Jini applications are generally deployed into multimachine environments, you will probably want to start multiple HTTP servers, one for each service.

How do you know whether or not code will need to be downloadable? This question will become clear as we delve into the Jini architecture. For now, you can assume that any core Jini component—such as the lookup service, the transaction manager, or JavaSpaces—may need to download code into other processes, and therefore their code should be accessible to at least one HTTP server running somewhere on your network.

Start the RMI Activation Daemon

Starting the RMI activation daemon couldn’t be easier. The executable for this process lives in \texttt{jdk1.3/bin/rmid}. If you’ve added this directory to your path, the most expedient way to start the activation daemon is by typing:

\begin{verbatim}
rmid -J-Dsun.rmi.activation.execPolicy=none
\end{verbatim}

The arguments on the command line cause the JDK 1.3 version of \texttt{rmid} to emulate the JDK 1.2 version security-wise. Essentially, it allows any program running on the machine full access to the facilities of \texttt{rmid}. With the tightening of security in the 1.3 platform, \texttt{rmid} no longer allows such access by default.

The security controls I’ve used here are acceptable on your own workstation or a “trusted” machine that you control access to, but is to general for a public server. For more details, read the manual pages for \texttt{rmid}, and see the information at:


An instance of the activation daemon must run on each host where activatable objects reside. This includes the Jini lookup service, the transaction manager, and JavaSpaces. And, of course, if you create any activatable objects yourself, you will need to run an instance of the activation daemon on the machines that host those objects.

The only other option you may consider passing to \texttt{rmid} is an explicit path to a log directory, by using the \texttt{-log <directory>} argument. The activation daemon needs access to a directory in which to write information about the services
Chapter 1  A New Computing Paradigm

that are registered with it. By default, it will simply create a directory called log underneath the current directory. So if you're not careful about where you start rmid, you may wind up with log directories scattered all over the system. You can use the -log argument to explicitly name a log directory that you can use each time you start rmid.

Start the Lookup Service

The Sun implementation of the lookup service is called “reggie,” and it lives in the reggie.jar file that comes with the distribution. Starting this service is a little more complicated than starting the others. For one thing, there are more arguments that you need to pass to the program to tell it how to behave.

The basic form of running the lookup service is as follows:

```
java -jar <reggie jar file>
   <lookup client codebase>
   <lookup policy file>
   <log directory>
   <lookup groups>
```

This is quite an eyeful, so I’ll walk through these options in a bit more detail. The first option here is an argument to the JVM itself, rather than to the Java program it will be running (reggie). The -jar option specifies that the main() routine for the program is contained in the specified JAR file; here, you should pass in the JAR file for the reggie service, reggie.jar, which lives in the lib directory of the Jini distribution.

Next are the arguments that are passed to the reggie program itself, rather than to the JVM that runs reggie. The next part of the command line is the codebase that specifies where code that will be downloaded to clients lives. The argument that you pass here should be a URL that points to the reggie-dl.jar file, which contains code used to access reggie that needs to be downloaded to callers. If you know for certain that all of your Jini services and applications will always run on the same machine, you can simply use a file: URL that points to the directory containing this JAR. As is more likely, though, you’ll be running code all over your network. In this case, you should pass a URL for the machine running your Web server that will be used to serve up code, for example, http:/hostname/reggie-dl.jar (replace “hostname” with the name of the machine that the Web server is running on, of course). You should make sure that the Web server can access the reggie-dl.jar file under its root directory.
Core Note: A word about codebase

There’s a certain art to figuring out how to set the codebase property which, unfortunately, eludes many newcomers to RMI and Jini. The codebase property is used as a way for the server—a program which supplies callable code to a client—to inform the client of where the code can be retrieved from.

When you set the codebase property on a program, you’re explicitly telling the server that, when it serializes an object for transmission to a client, it should “annotate” the serialized object with the codebase URL that says where to fetch the code for that object.

Serialization only transmits the member data within an object, not the code itself. Using codebase is a way for a server to tell a client how to download the code as well.

In general, an HTTP server will be running on each server machine that will export code. This HTTP server may be a separate process, such as you see here, or may be an extremely lightweight entity that lives within the server application itself.

There are two common mistakes to watch out for when using codebase. The first is that you should take great care when using file: URLs. File: URLs specify that the code for a service is accessible through the filesystem. This will work great as long as both the client and the server share the same filesystem. But if one of these components is moved to a different host, your once-working code will now break, since the common filesystem is no longer accessible.

A second precaution is to never use “localhost” in a codebase URL. The hostname “localhost” is a shortcut for referring to the host that is evaluating the hostname. If you specify localhost in a codebase, this means that clients will attempt to load code from their own machines, rather than the server’s machine. This is certainly not what you want, and the source of potentially hard-to-track bugs!

You should always use the name or the IP address of the host on which the HTTP service will run—which is usually the same host that the Jini service itself will run on. We’ll discuss codebase a bit more in the appendix on RMI.

Next you see a security policy file. Reggie, like many Jini services, registers itself with the RMI activation daemon. The activation daemon can start reggie “on demand,” and can even restart it if the service crashes. The security policy file you specify here is passed as an argument to the RMI activation subsystem so
that, whenever it activates a `reggie` process, it will begin running with the correct security permissions. Whenever you see a security policy file being passed on the command line like this (as opposed to being set via a property to the JVM), it is typically being used to set security for future activations of the service via the activation daemon.

The next option is a directory into which `reggie` will write its logs. You can pass any directory that is accessible on the filesystem where `reggie` is running; make sure you use an absolute pathname here. You should create any leading directories up to the actual log directory itself—although `reggie` will create its own log directory, it won’t create any missing directories earlier in the path. You’ll see many of the Jini services using log directories. These are used so that the services can checkpoint their state periodically, and recover after a crash. If you’re planning on leaving the services up for long periods, you’ll want to use a log directory that’s on a stable, persistent filesystem.

Finally, you pass a list of `groups` for which the lookup server will provide service. Groups are further discussed a bit later, but essentially they are names that can be used to group clusters of Jini services together. `Reggie` understands two “special” group names that have particular meanings. Passing “public” here indicates to use an unnamed public group in which, by convention, services should register by default; passing “none” means to use no group at all (which is not of much value to us). For now, just use “public” as the group.

Here’s a complete command line for starting `reggie`. Be sure, of course, to replace “hostname” and “port” with the name and port number you used to start the HTTP server in the earlier step.

*On Windows:*

```bash
java -jar c:jini1_1\lib\reggie.jar
   http://hostname:port/reggie-dl.jar
c:\jini1_1\example\lookup\policy.all
c:\temp\reggie_log
public
```

*On UNIX:*

```bash
java -jar /files/jini1_1/lib/reggie.jar
   http://hostname:port/reggie-dl.jar
   /files/jini1_1/example/lookup/policy.all
   /var/tmp/reggie_log
public
```

When you run the lookup service, you’ll notice that the command line returns to you after a few seconds (or perhaps longer if you’re running on a slow machine
or if there is a configuration problem). This is because the program that you’ve run has actually just registered `reggie` with the RMI activation daemon, so that it’ll be started by the activation daemon the first time it is needed; the above command doesn’t actually launch `reggie` itself. So the fact that the command returns after registration with the activation daemon is perfectly normal behavior.

You should be sure you understand the implications of the fact that `reggie` uses the activation daemon. First off, it means that this should be the only time you ever have to start `reggie` on the machine on which you just started it. The activation daemon will handle restarting it in the future, as long as (1) the activation daemon is running, and (2) the activation daemon has access to the log files it uses to save information about activatable programs. When you run `rmid`, it’ll create a directory called `log` under the directory that was current when you ran it. Later—say, when the machine reboots—you should be able to start `rmid` in this same directory. It will find its log files and thus be able to restart any activatable programs that it knew about during its last run, including `reggie`.

For its part, `reggie` should be able to find the log directory it was using to store persistent information when it restarts. So you should make sure that the `reggie_log` directory provided on the command line above is in a location that will be stable and accessible across reboots, if you want service registrations to be persistent across reboots.

Reggie is a complicated program, and its interactions with the activation daemon make it even more complicated for the people unfamiliar with this facility. If you’re having problems with `reggie`, you may want to take a look at the documentation and manual pages that come with the Jini distribution, or at [http://www.kedwards.com/jini/reggie.html](http://www.kedwards.com/jini/reggie.html), which has a more thorough overview of the `reggie` lifecycle and details on how to configure the program.
Listings 1–1 and 1–2 offer scripts for Windows and UNIX that you can use to start the lookup service; these scripts are available from the FTP site as mentioned in the introduction.

### Listing 1–1  lookup.bat Script for Windows

```batch
REM Set this to wherever Jini is installed
set JINI_HOME=C:\jini1_1
REM Set this to the host where the Web server runs
set HOSTNAME=hostname

REM Everything below should work with few changes
set POLICY=%JINI_HOME%\example\lookup\policy.all
set JARFILE=%JINI_HOME%\lib\reggie.jar
set CODEBASE=http://%HOSTNAME%:8080/reggie-dl.jar
set LOG_DIR=C:\temp\reggie_log
set GROUP=public

java -jar %JARFILE% %CODEBASE% %POLICY% %LOG_DIR% %GROUP%
```
Getting and Installing Jini

Running the Sample Program

At this point, you should have all of the required Jini services running on your machine, and you’re ready to make sure everything works. We’ll fire off a simple example—a browser application—that comes with the Jini distribution. To try out the browser, use the following command line:

**On Windows:**

```java
java -cp c:\jini1_1\lib\jini-examples.jar
   -Djava.security.policy=
       c:\jini1_1\example\browser\policy
   -Djava.rmi.server.codebase=
       http://hostname:8080/jini-examples-dl.jar
com.sun.jini.example.browser.Browser
```

**On UNIX:**

```java
java -cp /files/jini1_1/lib/jini-examples.jar
   -Djava.security.policy=
       /files/jini1_1/example/browser/policy
   -Djava.rmi.server.codebase=
       http://hostname:8080/jini-examples-dl.jar
com.sun.jini.example.browser.Browser
```

Listing 1–2 lookup.sh Script for UNIX

```bash
#!/bin/sh

# Set this to wherever Jini is installed
JINI_HOME=/files/jini1_1

# Set this to wherever the Web server is running
HOSTNAME=hostname

# Everything below should work with few changes
POLICY=$JINI_HOME/example/lookup/policy.all
JARFILE=$JINI_HOME/lib/reggie.jar
CODEBASE=http://$HOSTNAME:8080/reggie-dl.jar
LOG_DIR=/var/tmp/reggie_log
GROUP=public

java -jar $JARFILE $CODEBASE $POLICY $LOG_DIR $GROUP
```
Chapter 1 A New Computing Paradigm

Be sure, of course, to substitute the paths you used to install the Jini JAR files, and to use the correct URL for your Web server, if you changed any of the default configurations.

You’ll note that the browser requires its own codebase property—this is because the various Jini run-time services will actually download code that allows them to communicate with the browser as it runs. This code happens to be in the jini-examples.jar file, which is in the same directory as all of the rest of the Jini JAR files. So if you’ve already got an HTTP server that is exporting this directory, you can refer to this HTTP server in your codebase. Otherwise you’ll have to start a new one that exports the browser’s downloadable code.

At this point, you should see the browser application shown in Figure 1-7 running on your screen. This application lets you look at the Jini services running on your network, control them, and change the attributes associated with each of them. The display should indicate that one or more “registrars” (lookup services) are available. If you do not see this, then the most likely problem is that you’ve misconfigured your codebase for the lookup service.

You can leave this application running to monitor the status of any Jini lookup services running. For now, most of the controls in this application may not make much sense to you—we’ll soon learn about Jini services and attributes, though. For now, if you see at least one Jini lookup service running—the one you just started—then things are set up properly.

![Figure 1–7](image-url)
Further Reading and Resources

If you want to read more about the early history of the Java project (as well as see a photo of the Star-7 device), you can check out James Gosling’s home page, which has an excellent introduction to the history of the project:

http://java.sun.com/people/jag/green/index.html

If you’re curious about the integration of nondesktop computers into our daily lives, and how computers can be made to “disappear” into the fabric of our surroundings, you may want to check out Mark Weiser’s vision of ubiquitous computing:

http://ubiq.com/hypertext/weiser/UbiHome.html

The New York Times article that broke the scoop on Jini still makes for some interesting reading:


The Wired interview with Bill Joy talks about Jini and the future visions of computing that inspired the Jini project:


I’ve tried to keep track of some of the most common problems (and their solutions!) that arise when developing with Jini. These are collected on a troubleshooting page that I keep at my web site. If you have trouble getting services to run, you might visit the page to see if you recognize the problem you’re having.

http://www.kedwards.com/jini

An excellent resource for Jini developers is the JINI-USERS mailing list. To subscribe, send mail to listserv@java.sun.com; the body of your message should contain the line subscribe jini-users. To unsubscribe, send a message to the same address, with the body containing signoff jini-users.

The RMI-USERS mailing list is also a great resource for getting help with RMI-related questions. To sign up, send a note to listserv@java.sun.com containing subscribe rmi-users. To sign off, send a note to the same address containing signoff rmi-users.
Finally, Sun has a Web site for the use of the larger Jini community. This Web site is a good place to check for code donated by community members, news, and upcoming Jini events. Registration is required, but grants you access to code from other community members, and gives you a space in which to upload your own code.

http://www.jini.org