CHAPTER 7
Leasing

IN DISTRIBUTED APPLICATIONS, THERE MAY BE partial failures of the network or of components on the network. Leasing is a way for components to register that they are alive, but to ensure that they are "timed out" if they fail or are unreachable. Leasing is the mechanism used between applications to give access to resources over a period of time in an agreed manner.

Leases are requested for periods of time, and these requests may be granted, modified, or denied. The most common example of a lease is when a service is registered with lookup services. A lookup service will not want to keep a service forever, because it may disappear. Keeping information about nonexistent services is a waste of resources on the lookup service and also may lead to clients wasting time trying to access services that aren't there. As a result, a lookup service will grant a lease saying that it will only keep information for a certain period of time, and the service can renew the lease later if it wants to.

Requesting and Receiving Leases

Leases are requested for a period of time. In Jini, a common use of leasing is for a service provider to request that a copy of the service be kept on a lookup service for a certain length of time, for delivery to clients on request. The service provider requests a time in the `ServiceRegistrar`'s `register()` method. Two special values of the time are

- `Lease.ANY`–the service lets the lookup service decide on the time
- `Lease.FOREVER`–the request is for a lease that never expires

The lookup service acts as the granter of the lease and decides how long it will actually create the lease for. (The lookup service from Sun typically sets the lease time as only five minutes.) Once it has done that, it will attempt to ensure that the request is honored for that period of time. The lease is returned to the service and is accessible through the `getLease()` method of the `ServiceRegistration` object.

These objects are shown in Figure 7-1. The following are typical calls to register the service and then find the lease:

```
ServiceRegistration reg = registrar.register();
Lease lease = reg.getLease();
```

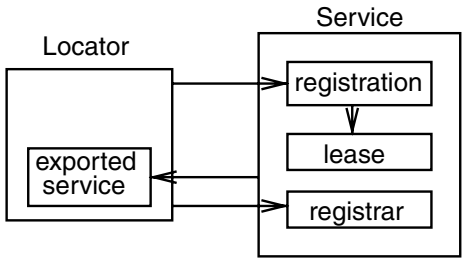

Figure 7-1. Objects in a leased system

The principal methods of the `Lease` object are these:

```
package net.jini.core;

public interface Lease {
    void cancel() throws
                  UnknownLeaseException,
                  java.rmi.RemoteException;
    long getExpiration();
    void renew(long duration) throws
                  LeaseDeniedException,
                  UnknownLeaseException,
                  java.rmi.RemoteException;
}
```

The expiration value returned from `getExpiration()` is the time in milliseconds since the beginning of the epoch (the same as in `System.currentTimeMillis()`). To find the amount of time still remaining from the present, the current time can be subtracted from this, as follows:

```
long duration = lease.getExpiration() - System.currentTimeMillis();
```

Cancellation

A service can cancel its lease by using `cancel()`. The lease communicates back to the lease management system on the lookup service, which cancels storage of the service.

Expiration

When a lease expires, it does so silently. That is, the lease granter (the lookup service) will not inform the lease holder (the service) that it has expired. While it might seem nice to get warning of a lease expiring so that it can be renewed, this would have to be done in advance of the expiration ("I'm just about to expire; please renew me quickly!") but this would complicate the leasing system and not be completely reliable anyway (for example, how far in advance is soon enough?).

Instead, it is up to the service provider to call `renew()` before the lease expires if it wishes the lease to continue. The parameter for `renew()` is in milliseconds, and represents an extra duration from now. This is in contrast to the expiration time returned from `getExpiration()`, which is measured since the epoch.

Renewing Leases

Jini supplies a `LeaseRenewalManager` class that looks after the process of calling `renew()` at suitable times.

> **WARNING** *In Jini 1.0, this class was in package* `com.sun.jini`*; in Jini 1.1 it is now in package* `net.jini.lease`.

```
package net.jini.lease;

public class LeaseRenewalManager {
    public LeaseRenewalManager();
    public LeaseRenewalManager(Lease lease,
                               long expiration,
                               LeaseListener listener);
    public void renewFor(Lease lease,
                         long duration,
                         LeaseListener listener);
```

```
public void renewUntil(Lease lease,
                       long expiration,
                       LeaseListener listener);
```

The `LeaseRenewalManager` manages a set of leases, which may be set by a constructor or added later by `renewFor()` or `renewUntil()`. The time requested in these methods is in milliseconds. The `expiration` time is measured since the epoch, whereas the `duration` time is measured from now.

Generally leases will be renewed and the manager will function quietly. However, the lookup service may decide not to renew a lease and will cause an exception to be thrown. This will be caught by the renewal manager and will cause the listener's `notify()` method to be called with a `LeaseRenewalEvent` as parameter, which will allow the application to take corrective action if its lease is denied. If the listener is `null`, then no notification will take place.

If you are using Jini 1.0, you have to be careful about setting the `duration` in `renewFor()` due to a bug that has since been fixed. If you want the service to be registered forever, it is tempting to use `Lease.FOREVER`. However, the Jini 1.0 implementation just adds this to `System.currentTimeMillis()`, which overflows to a negative value that is not checked. As a result, it never does any renewals. You need to check that

```
duration + System.currentTimeMillis() > 0
```

before calling `renewFor()`. This is fixed in Jini 1.1. The `renewUntil()` method can use `Lease.FOREVER` with no problems.

Granting and Handling Leases

The preceding discussion looked at leases from the side of the client that receives a lease and has to manage it. The converse of this is the agent that grants leases and has to manage things from its side. This is more advanced material that you can skip for now if you want—it is not really needed until Chapter 14. An example of creating a lease is also given in Chapter 13.

A lease can be granted for almost any remote service—any one where one object wants to maintain information about another one that is not within the same virtual machine. As with other remote services, there are the added partial failure modes, such as network crash, remote service crash, timeouts, and so on. An object that keeps information on a remote service will hand out a lease to the service and will want the remote service to keep "pinging" it periodically to say that it is still alive and that it wants the information kept. Without this periodic assurance, the object might conclude that the remote service has vanished or is somehow unreachable, and that it should discard the information about it.

Leases are a very general mechanism for allowing one service to have confidence in the existence of the other for a limited period. Because they are general, they allow for a great deal of flexibility in use. Because of the potential variety of services, some parts of the Jini lease mechanism cannot be completely defined and must be left as interfaces for applications to fill in. This generality means that all of the details are not filled in for you, as your own requirements cannot be completely predicted in advance.

A lease is given as an interface, and any agent that wishes to grant leases must implement this interface. Jini provides two implementations, an `AbstractLease` and a subclass of this, a `LandlordLease`.

A main issue in implementing a particular lease class lies in setting a policy for handling the initial request for a lease period and in deciding what to do when a renewal request comes in. A couple of simple possibilities are these:

- Always grant the requested time
- Ignore the requested time and always grant a fixed time

Of course, there are many more possibilities based on the lessor's expected time to live, system load, etc.

There are other issues, though. Any particular lease will need a time-out mechanism. Also, a group of leases can be managed together, and this can reduce the amount of overhead involved in managing individual leases.

Abstract Lease

An abstract lease gives a basic implementation of a lease that can almost be used for simple leases.

> **WARNING** *This class, and those that depend on it, are still not fully specified and may change in future versions of Jini.*

```
package com.sun.jini.lease;

public abstract class AbstractLease implements Lease, java.io.Serializable {
    protected AbstractLease(long expiration);
    public long getExpiration();
    public int getSerialFormat();
```

```
    public void setSerialFormat(int format);
    public void renew(long duration);
    protected abstract long doRenew(long duration);
}
```

This class supplies straightforward implementations of much of the `Lease` interface, with three provisos:

- The constructor is `protected`, so that constructing a lease with a specified duration is devolved to a subclass. This means that a lease duration policy must be set by this subclass.
- The `renew()` method calls into the abstract `doRenew()` method, again to force a subclass to implement a renewal policy.
- The `Lease` interface does not implement the `cancel()` method, so this must also be left to a subclass.

Thus, this class implements the easy things, and leaves all matters of policy to concrete subclasses.

Landlord Lease Package

The *landlord* is a package that allows more complex leasing systems to be built. It is not part of the Jini specification, but is supplied as a set of classes and interfaces. The set is not complete in itself—some parts are left as interfaces and need to have class implementations. These will be supplied by a particular application.

WARNING *This landlord package is part of the* `com.sun.jini` *package, which may change for Jini 1.1.*

A landlord looks after a set of leases. Leases are identified to the landlord by a *cookie*, which is just some object that uniquely labels each lease to its landlord. It could be an `Integer`, for example, with a new value for each lease. A landlord does not need to create leases itself, as it can use a landlord lease factory to do this. (But, of course, it can create them, depending on how an implementation is done.) When a client wishes to cancel or renew a lease, it asks the lease to perform the renewal, and in turn the lease asks its landlord to do it. A client is unlikely to ask the landlord directly, as it will only have been given a lease, not a landlord.

The principal classes and interfaces in the `landlord` package are shown in Figure 7-2, where the interfaces are shown in italicized font and the classes in normal font.

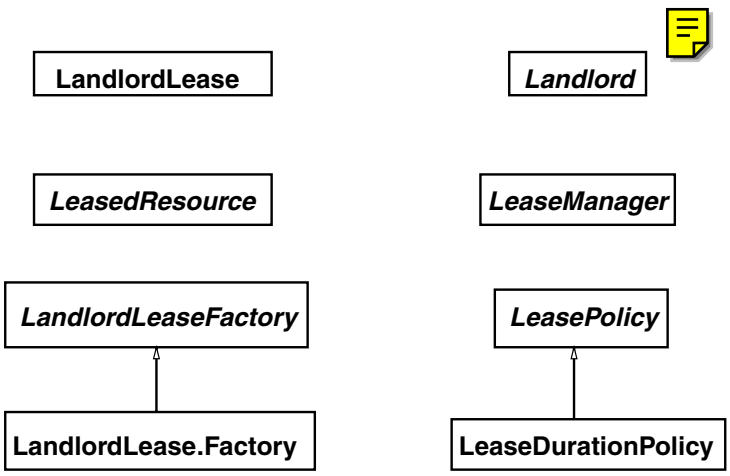

Figure 7-2. Class diagram of the landlord package

This fairly complex set of classes and interfaces is driven by a number of factors:

- The key object in a landlord system is the landlord itself. Because there are many ways that a landlord could manage a set of leases, the `Landlord` is an interface rather than a class, with many possible implementations.

- Because there are many possible landlords, there could be many possible lease-types created, which will all be subclasses of `Lease`. A common design pattern in such a circumstance is to use a *factory* object to create the leases. These factory objects will implement the `LandlordLeaseFactory` interface.

- A simple lease implementation was needed for a variety of situations, and this is the `LandlordLease` class. When a particular implementation is chosen, the factory pattern says that a new factory is needed to create new objects. So to create `LandlordLease` objects, the `LandlordLease.Factory` factory class is used. (Note the dot (`.`) in the `LandlordLease.Factory` class name, which distinguishes it from the `LandlordLeaseFactory` interface.) A lease (on the client) also requires the existence of some handler for its methods on the lease-granting side, which is the landlord.

- To handle all policy issues, such as initial granting of lease times, and requests for lease renewal, a policy object is used. There can be many possible policies

implementing the `LeasePolicy` interface. Each lease policy needs to make decisions about leases, but it needs to make decisions on the lease-granting side, so a lease policy needs to keep enough information locally to make proper decisions. The information about leases on the granting side is kept in *leased resources*, which are implementations of the `LeasedResource` interface.

- For each lease on the client side, there will be a leased resource on the granting side. These must be stored and managed somehow. There may be only a few leases, but there could be many thousands. There could be relationships between them (such as linear order), or none at all. So, to avoid decisions about storage structures that would be wrong half of the time, lease management is just left as an interface.

Java uses interfaces as specifications without implementation details. For individual classes this is often fine. However, using interfaces can be limiting when you are dealing with a set of classes that are expected to interact in certain ways. Interfaces do not show the interactions that may need to exist in order for an implementation of the set of classes to function together. This means the interface definitions are not complete as they stand, because they fail to show the links between classes that must exist in any implementation. To see what these links actually are, let us look at a simple implementation for the `Foo` landlord package.

If we have a landlord for a `Foo` resource, then we could end up with the class structure shown in Figure 7-3.

This diagram uses a UML class diagram annotated with arrows and multiplicities. An association with an arrow means that the object at the source of the arrow will know about the object at the other end of the arrow. For example, each `LandlordLease` knows about (has a reference to) a `FooLandlord`, but the landlord does not know about any leases. At each end of each association between classes, the multiplicity of that end of the link is also shown. A "*" is a wildcard pattern, meaning "zero to many." So for example, any number of `LandlordLeases` (from zero upwards) may know about a single `FooLandlord`.

Some comments are appropriate about the directions and multiplicities:

- A landlord can be managing many leases, but it doesn't know what the leases are—the leases know their landlord, and they call its methods using the lease cookie. So many `LandlordLease` objects contain a reference to a `FooLandlord`.

- Certain requests need to be forwarded through the system. For example, a `renew()` request from a lease will get passed to a landlord. The landlord cannot handle it directly, since the renewal is a matter requiring policy decisions. It must be passed to a lease policy object. One way of doing this (as shown in Figure 7-3) is for the landlord to have a reference to a lease

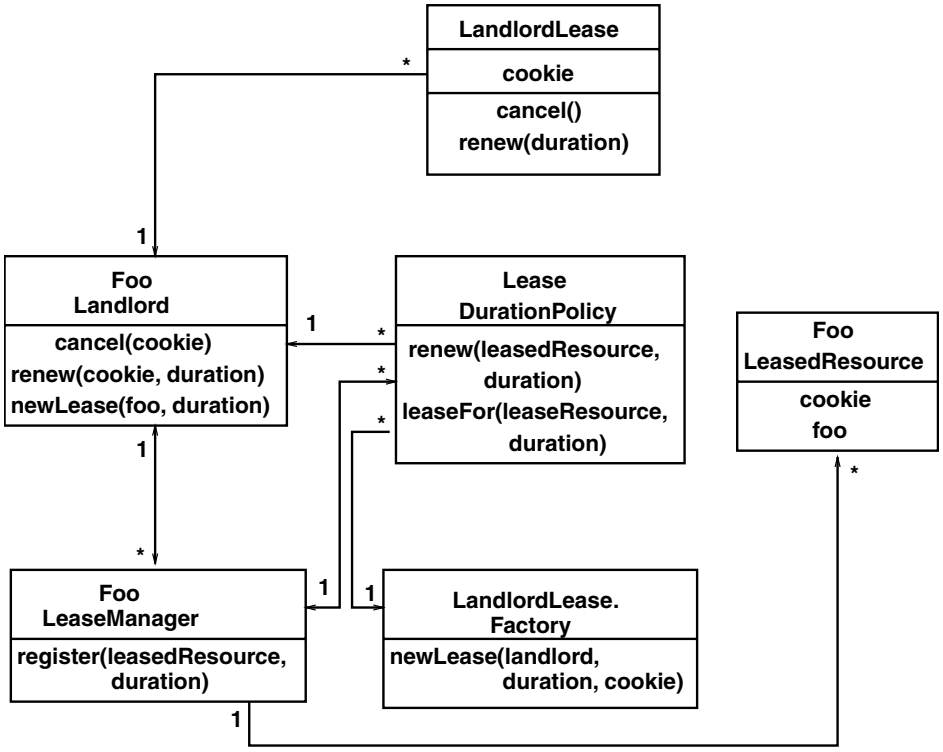

Figure 7-3. Class diagram of a landlord implementation

manager, which has a reference to a lease policy. Similarly, a `newLease()` request from the landlord will need to invoke a `newLease()` method on the factory, and this can be done by ensuring that the lease policy also has a reference to the factory.

- A factory may be used by many lease policies, a policy may be used by many lease managers, and a lease manager may be used by many landlords.

LandlordLease Class

The `LandlordLease` class extends `AbstractLease`. This class has the private fields `cookie` and `landlord`, as shown in Figure 7-4.

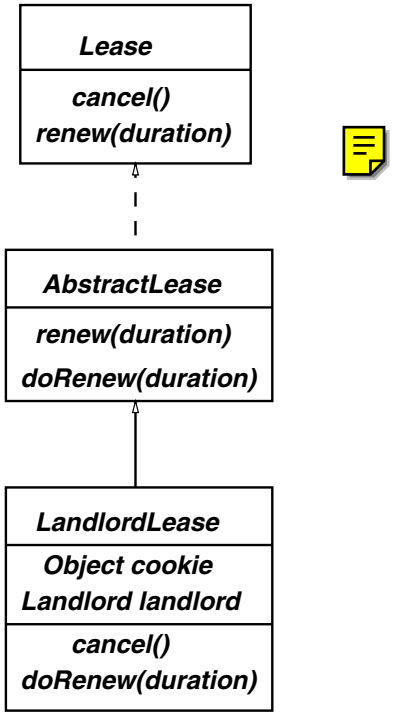

Figure 7-4. The class diagram for LandlordLease

Implementation of the methods `cancel()`and `doRenew()` in the `LandlordLease` is deferred to its landlord. The implementation of these methods in the `LandlordLease` simply passes the requests on to the landlord:

```
public void cancel() {
    landlord.cancel(cookie);
}

protected long doRenew(long renewDuration) {
    return landlord.renew(cookie, renewDuration);
}
```

The `LandlordLease` class can be used as is, with no subclassing needed. Note that the landlord system produces these leases but does not actually keep them anywhere—they are passed on to clients, which then use the lease to call the landlord and hence interact with the landlord lease system. Within the landlord system, on the lessor side, the cookie is used as an identifier for the lease.

LeasedResource Interface

A `LeasedResource` is a convenience wrapper around a resource that includes extra information about a lease and methods for use by landlords. It defines an interface as follows:

```
public interface LeasedResource {
    public void setExpiration(long newExpiration);
    public long getExpiration();
    public Object getCookie();
}
```

This interface includes the *cookie*, a unique identifier for a lease within a landlord system, as well as expiration information for the lease. This is all the information maintained about the lease that has been given out to a client.

An implementation of `LeasedResource` will typically include the resource that is leased, plus a method of setting the cookie. The following code shows an example:

```
/**
 * FooLeasedResource.java
 */
package foolandlord;

import com.sun.jini.lease.landlord.LeasedResource;

public class FooLeasedResource implements LeasedResource  {

    static protected int cookie = 0;
    protected int thisCookie;
    protected Foo foo;
    protected long expiration = 0;

    public FooLeasedResource(Foo foo) {
        this.foo = foo;
        thisCookie = cookie++;
    }

    public void setExpiration(long newExpiration) {
        this.expiration = newExpiration;
    }

    public long getExpiration() {
        return expiration;
    }
```

```
    public Object getCookie() {
        return new Integer(thisCookie);
    }

    public Foo getFoo() {
        return foo;
    }
} // FooLeasedResource
```

LeasePolicy Interface

A lease policy is used when a lease is first granted and when it tries to renew itself. The time requested may be granted, modified, or denied. A lease policy is specified by the `LeasePolicy` interface.

```
package com.sun.jini.lease.landlord;

public interface LeasePolicy {

    public Lease leaseFor(LeasedResource resource, long requestedDuration)
        throws LeaseDeniedException;

    public long renew(LeasedResource resource, long requestedDuration)
        throws LeaseDeniedException, UnknownLeaseException;

    public boolean ensureCurrent(LeasedResource resource);
}
```

This interface includes a factory method, `leaseFor()`, that returns a lease based on the policy and request.

LeaseDurationPolicy Class

An implementation of the `LeasePolicy` interface is given by the `LeaseDurationPolicy` class. This class grants and renews leases based on constant values for maximum and default lease durations, as shown here:

```
package com.sun.jini.lease.landlord;

public class LeaseDurationPolicy implements LeasePolicy {

    public LeaseDurationPolicy(long maximum, long defaultLength,
```

```
        Landlord landlord, LeaseManager mgr, LandlordLeaseFactory factory);

    public Lease leaseFor(LeasedResource resource, long requestedDuration)
        throws LeaseDeniedException;
    public long renew(LeasedResource resource, long requestedDuration);
    public boolean ensureCurrent(LeasedResource resource);
}
```

In addition to implementing the interface methods, the constructor also passes in the factory to be used (which will probably be a `LandlordLease.Factory`) and maximum and default lengths for leases. The maximum duration is to set a hard upper limit (which could be, say, `Lease.FOREVER`), while the default is what is granted if the client asks for a duration of `Lease.ANY`.

LeaseManager Interface

The operations that can be carried out on a lease are creation, renewal, and cancellation. The first two are subject to the lease policy and must be handled by the `leaseFor()` and `renew()` methods of the policy. These set or alter the properties of a single lease. There may be many leases for a resource, or even many resources with one or more leases. Some level of management for a group of leases may be needed, and this is done by a `LeaseManager`.

The `LeaseManager` interface is defined as follows:

```
package com.sun.jini.lease.landlord;

public interface LeaseManager {
    public void register(LeasedResource resource, long duration);
    public void renewed(LeasedResource resource, long duration,
                        long oldExpiration);
}
```

This `LeaseManager` doesn't actually manage the leases, since they have been given to the client. Rather, it handles the lease resource, which has the cookie identifier and the expiration time for the lease.

An implementation of `LeaseManager` will look after a set of leases (really, their resources) by adding a new lease resource to its set for each lease, and by updating information about renewals. The interface does not include a method for informing the manager of cancelled leases, though—that is done to the `Landlord` instead, by the lease when the lease's `cancel()` method is called.

This split responsibility between `LeaseManager` and `Landlord` is a little awkward and can possibly lead to memory leaks, with the manager holding a reference to a

lease (resource) that the landlord has cancelled. Either the list of lease resources must be shared between the two, or the landlord must ensure that it passes on cancellations to the manager.

There is also the question of how the lease manager is informed of changes to individual leases by the lease policy. The `LeaseDurationPolicy` will pass on this information in its `leaseFor()` and `renew()` methods, but other implementations of `LeasePolicy` need not. As we only use the `LeasePolicy` implementation, we are okay here.

A third question is who looks after leases expiring, and how this can be done. No part of the landlord specifications talk about this or give a suitable method. This suggests that it, too, is subject to some sort of policy, but it is not one with landlord support. It is left to implementations of one of the landlord interfaces, or to a subclass. A convenient place to locate this checking is in the lease manager, because it has knowledge of all the leases and their duration. Possible ways of doing this include the following:

- A thread per lease, which will sleep and time out when the lease should expire. This will need to sleep again if the lease has been renewed in the meantime.

- A single sleeper thread sleeping for the minimum period of all leases. This may need to be interrupted if a new lease is created with a shorter expiration period.

- A polling mechanism in which a thread sleeps for a fixed time and then cleans up all leases that have expired in the meantime.

- A lazy method, in which no active thread looks for lease expiries but just cleans them up if it comes across expired leases while doing something else. (This lazy approach is taken by the JavaSpaces Outrigger service, which grants leases for Entry objects).

The `FooLeaseManager` implements this third polling mechanism method:

```
/**
 * FooLeaseManager.java
 */
package foolandlord;

import java.util.*;
import net.jini.core.lease.Lease;
import com.sun.jini.lease.landlord.LeaseManager;
import com.sun.jini.lease.landlord.LeasedResource;
import com.sun.jini.lease.landlord.LeaseDurationPolicy;
```

```
import com.sun.jini.lease.landlord.Landlord;
import com.sun.jini.lease.landlord.LandlordLease;
import com.sun.jini.lease.landlord.LeasePolicy;

public class FooLeaseManager implements LeaseManager {

    protected static long DEFAULT_TIME = 30*1000L;

    protected Vector fooResources = new Vector();
    protected LeaseDurationPolicy policy;

    public FooLeaseManager(Landlord landlord) {
        policy = new LeaseDurationPolicy(Lease.FOREVER,
                                         DEFAULT_TIME,
                                         landlord,
                                         this,
                                         new LandlordLease.Factory());
        new LeaseReaper().run();
    }

    public void register(LeasedResource r,long duration) {
        fooResources.add(r);
    }

    public void renewed(LeasedResource r, long duration, long olddur) {
        // no smarts in the scheduling, so do nothing
    }

    public void cancelAll(Object[] cookies) {
        for (int n = cookies.length; --n >= 0; ) {
            cancel(cookies[n]);
        }
    }

    public void cancel(Object cookie) {
        for (int n = fooResources.size(); --n >= 0; ) {
            FooLeasedResource r = (FooLeasedResource) fooResources.elementAt(n);
            if (r.getCookie().equals(cookie)) {
                fooResources.removeElementAt(n);
            }
        }
    }

    public LeasePolicy getPolicy() {
```

```
            return policy;
        }

        public LeasedResource getResource(Object cookie) {
            for (int n = fooResources.size(); --n >= 0; ) {
                FooLeasedResource r = (FooLeasedResource) fooResources.elementAt(n);
                if (r.getCookie().equals(cookie)) {
                    return r;
                }
            }
            return null;
        }

        class LeaseReaper extends Thread {
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(DEFAULT_TIME) ;
                    }
                    catch (InterruptedException e) {
                    }
                    for (int n = fooResources.size()-1; n >= 0; n--) {
                        FooLeasedResource r = (FooLeasedResource)
                                              fooResources.elementAt(n)
    ;
                        if (!policy.ensureCurrent(r)) {
                            System.out.println("Lease expired for cookie = " +
                                               r.getCookie()) ;
                            fooResources.removeElementAt(n);
                            // replace this landlord.cancel(r.getCook    ) ;

                        }
                    }
                }
            }
        }

    } // FooLeaseManager
```

Landlord Interface

The `Landlord` is the final interface in the package that we need for a basic landlord system. Other classes and interfaces, such as `LeaseMap` are for handling leases in batches, and will not be dealt with here. The `Landlord` interface is as follows:

```
package com.sun.jini.lease.landlord;

public interface Landlord extends Remote {

    public long renew(Object cookie, long extension)
        throws LeaseDeniedException, UnknownLeaseException, RemoteException;

    public void cancel(Object cookie)
        throws UnknownLeaseException, RemoteException;

    public RenewResults renewAll(Object[] cookie, long[] extension)
        throws RemoteException;

    public void cancelAll(Object[] cookie)
        throws LeaseMapException, RemoteException;
}
```

The `renew()` and `cancel()` methods are usually called from the `renew()` and `cancel()` methods of a particular lease. The `renew()` method needs to use a policy object to ask for renewal, and in the `FooLandlord` implementation, it gets this policy from the `FooLeaseManager`. The `cancel()` method needs to modify the list of leases, and in the `FooLandlord` implementation, it passes this on to the `FooLeaseManager`, since that is the only object that maintains a list of resources.

There must be a method to ask for a new lease for a resource, and this is not specified by the landlord package. This request will probably be made on the lease-granting side, and this should have access to the landlord object, which forms a central point for lease management. So, an implementation of this interface will quite likely have a method such as

```
public Lease newFooLease(Foo foo, long duration);
```

which will give a lease for a resource.

The lease used in the landlord package is a `LandlordLease`. This contains a private field, which is a reference to the landlord itself. The lease is given to a client as a result of `newFooLease()`, and this client will usually be a remote object. This will involve serializing the lease and sending it to this remote client. While serializing it, the landlord field will also be serialized and sent to the client.

When the client methods such as `renew()` are called, the implementation of the `LandlordLease` will make a call to the landlord. The lease is on the client, which by then will be remote from its origin where the landlord lives. That means the landlord object invoked by the lease will need to be a remote object making a remote call. The `Landlord` interface already extends `Remote`, but if it is to run as a remote object, then the easiest way is for `FooLandlord` to extend the `UnicastRemoteObject` class.

Putting all this together for the `FooLandlord` class gives us this:

```
/**
 * FooLandlord.java
 */

package foolandlord;

import com.sun.jini.lease.landlord.*;
import net.jini.core.lease.LeaseDeniedException;
import net.jini.core.lease.Lease;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;

public class FooLandlord extends  UnicastRemoteObject
                          implements Landlord {

    FooLeaseManager manager;

    public FooLandlord() throws java.rmi.RemoteException {
        manager = new FooLeaseManager(this);
    }

    public void cancel(Object cookie) {
        manager.cancel(cookie);
    }

    public void cancelAll(Object[] cookies) {
        manager.cancelAll(cookies);
    }

    public long renew(java.lang.Object cookie,
                      long extension)
        throws net.jini.core.lease.LeaseDeniedException,
               net.jini.core.lease.UnknownLeaseException {
        LeasedResource resource = manager.getResource(cookie);
        if (resource != null) {
```

```
            return manager.getPolicy().renew(resource, extension);
        }
        return -1;
    }

    public Lease newFooLease(Foo foo, long duration)
        throws LeaseDeniedException {
        FooLeasedResource r = new FooLeasedResource(foo);
        return manager.getPolicy().leaseFor(r, duration);
    }

    public Landlord.RenewResults renewAll(java.lang.Object[] cookies,
                                          long[] extensions) {
        long[] granted = new long[cookies.length];
        Exception[] denied = new Exception[cookies.length];

        for (int n = cookies.length; --n >= 0; ) {
            try {
                granted[n] = renew(cookies[n], extensions[n]);
                denied[n] = null;
            } catch(Exception e) {
                granted[n] = -1;
                denied[n] = e;
            }
        }
        return new Landlord.RenewResults(granted, denied);
    }
} // FooLandlord
```

Building an implementation of the landlord package, such as the Foo package, means providing implementations of the `Landlord`, `LeasedResource`, and `LeaseManager` interfaces. This has been done using the `FooLandlord`, `FooLeasedResource`, and `FooLeaseManager` classes.

Summary

Leasing allows resources to be managed without complex garbage-collection mechanisms. Leases received from services can be dealt with easily, using `LeaseRenewalManager`. Entities that need to hand out leases can use a system, such as the landlord system, to handle these leases.