

# Runtime Client Services

CHAPTER

**11**

## IN THIS CHAPTER

- Introduction 320
- File Management 327
- Resource Management 332
- Persistence 333
- Controlling Installations 338
- Other Services 342
- An Example Application 343
- Summary 347

In this chapter, we will examine the runtime services offered by a JNLP Client to its launched applications.

## Introduction

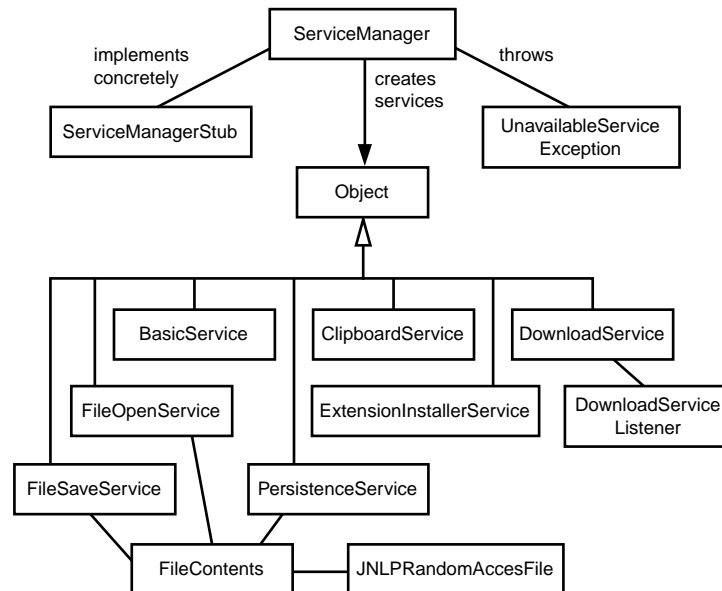
To compile the code in this chapter, you need to add the `javax.jnlp` library (released from Sun) in your classpath. It is freely available through the Java Developer Connection or more quickly from the Java Web Start product page at <http://www.javasoft.com/products/javaWebstart>.

The services offered by the JNLP Client to launched applications are always invoked by obtaining a service object from the `javax.jnlp.ServiceManager` class (as shown in the following code):

```
service =
(ExtensionInstallerService)ServiceManager.lookup("javax.jnlp.ExtensionInstallerService");
```

Once obtained, the particular service object clients can use it as needed. We won't discuss the API details here (have a look at Appendix B, "The JNLP Specification," or at the `jnlp` package documentation); instead, we will give practical code that is useful in real-world situations.

Figure 11.1 depicts a UML class diagram showing the `javax.jnlp` package classes.



**FIGURE 11.1**

*The class diagram for the JNLP classes.*

## The ServiceManager class

The `ServiceManager` class provides the means for obtaining services from the JNLP Client through the use of the `lookup()` method. This method is called by passing it a string with the fully qualified class name. To discover all the available services, use the method `getServiceNames`. The method `setServiceManagerStub` is used only by the JNLP Client to install its implementation of the services factory.

### NOTE

The superclass of all the service classes is `Object` (see Figure 11.1).

## The BasicService class

To begin with, we will study the `BasicService` class, the most general-purpose service of all the JNLP services. It provides four methods:

- `getCodebase()`. Returns the codebase URL as specified in the main JNLP file.
- `isOffline()`. Useful for applications or installers that need to determine whether the client computer is currently connected.
- `showDocument()`. Launches the installed Web browser to the specified URL. Used, for example, in the `JNLPLabel` class in this chapter.
- `isWebBrowserSupported()`. Useful to know whether a Web browser is currently installed on the given OS.

## A Utility class

We already saw the `Utilities` class in the previous chapter. This class helped us keep the code short and meaningful. It is a singleton (see note below), with two instantiation methods, namely `getInstance()` and `getInstance(String)`. The latter initializes the class with a specified resource bundle for dealing with multilingual applications (but it is always a good idea to use it). These initialization methods are not required. If we call a static method without prior initialization:

```
Utilities.getService("javax.jnlp.ClipboardService");
```

The `Utilities` class is able to initialize itself transparently.

**NOTE**

A singleton class is a class that can have only one instance, reachable from a global point of access. There are several slightly different ways to achieve this behavior with the Java language, but a private constructor is always needed.

Listing 11.1 illustrates the various aspects of the `javax.jnlp` API as it pertains to the client side.

**LISTING 11.1** The Utilities Class

```
package com.marinilli.b2.c11.util;
import javax.swing.ImageIcon;
import java.awt.image.BufferedImage;
import java.net.URL;
import javax.jnlp.UnavailableServiceException;
import javax.jnlp.ServiceManager;
import javax.jnlp.BasicService;
import java.util.ResourceBundle;
/**
 * Chapter 11 - Utilities
 * @author Mauro Marinilli
 * @version 1.0
 */
public class Utilities {
    private static Utilities util;
    private static ClassLoader loader;
    private static BasicService basicService;
    private static ImageIcon EMPTY_ICON =
        new ImageIcon(new BufferedImage(24,24,BufferedImage.TYPE_INT_RGB));
    private ResourceBundle msg;
    private static String bundleFilename = "messages";
    private Utilities() {
        loader = getClass().getClassLoader();
        basicService =
            (BasicService)getService("javax.jnlp.BasicService");
        initializeDefaultResources();
        try {
            msg = ResourceBundle.getBundle(bundleFilename);
        } catch (java.util.MissingResourceException mre) {
            msg = null;
            System.out.println("Utilities (init)
↳ Couldn't find any resource bundle.");
```

**LISTING 11.1** Continued

```
    }
}
private void initializeDefaultResources() {
    //initializes the empty icon
    java.awt.Graphics g = EMPTY_ICON.getImage().getGraphics();
    g.drawLine(8,8,16,16);
    g.drawLine(8,16,16,8);
}
public static ClassLoader getClassLoader(){
    if (util==null)
        getInstance();
    return loader;
}
public static ImageIcon getImageIcon(String name){
    if (util==null)
        getInstance();
    URL res = loader.getResource(name);
    if (res!=null)
        return new ImageIcon(res);
    return EMPTY_ICON;
}
public static Object getService(String fullyQName){
    Object service = null;
    try {
        service = ServiceManager.lookup(fullyQName);
    } catch (UnavailableServiceException use) {
        System.out.println(util.getClass()+"getService("+ fullyQName +") "+use);
    }
    return service;
}
public static BasicService getBasicService(){
    if (util==null)
        getInstance();
    return basicService;
}
public static String getMsg(String key){
    if (util==null)
        getInstance();
    if (util.msg!=null) {
        String m = util.msg.getString(key);
        if (m!=null)
            return m;
    }
    return "";
}
```

**11**RUNTIME CLIENT  
SERVICES

**LISTING 11.1** Continued

---

```
}
public static Utilities getInstance(){
    if (util==null)
        util = new Utilities();
    return util;
}
public static Utilities getInstance(String resourceName){
    bundleFilename = resourceName;
    if (util==null)
        getInstance();
    else
        System.out.println(
            util.getClass()+" getInstance must be invoked as initializer.");
    return util;
}
}
```

---

This class provides several services added on top of the JNLP API:

- Robust resources retrieval. For icons and resource bundles, if the given item is not found, a default value is always returned. For messages, an empty string is returned in case of a missing value.
- For icons, a default icon is always provided by creating it in code, so that `NullPointerException` messages for inexistent icons are never thrown. See lines 18–19 and the `initializeDefaultResources` method at lines 34–38.
- It saves programmers from lengthy try-catch constructs in standard services creation code. Note also that an instance of the `BasicService` class is always created because it is usually the most often used service.

The `Utilities` class has been kept simple to serve a didactic purpose as well. You can always expand it as needed.

## A Hypertext Swing Label Class

To see a simple and interesting example of how it is possible to take advantage of the JNLP Client API and the `Utilities` class, let's use a very simple `JLabel` component that can be clicked just like a hyperlink in a Web page. When this action is completed, a Web browser will open, pointing to the given URL. Listing 11.2 shows the code.

**LISTING 11.2** The JNLPLabel Class

```
package com.marinilli.b2.c11.util;
import javax.swing.JLabel;
import java.awt.event.*;
import java.util.*;
import java.awt.Cursor;
import java.net.URL;
import javax.swing.event.*;
/**
 * Chapter 11 - A Label that launches a browser
 * @author Mauro Marinilli
 * @version 1.0
 */
public class JNLPLabel extends JLabel implements MouseListener {
    private String text = "";
    private URL url= null;
    private static final String COLOR_BEFORE_CLICKING = "blue";
    private static final String COLOR_AFTER_CLICKING = "purple";
    private boolean alreadyClicked = false;
    private transient Vector hListeners;

    public JNLPLabel() {
        addMouseListener(this);
    }
    public void mouseClicked(MouseEvent e) {
        alreadyClicked=true;
        java.awt.Toolkit.getDefaultToolkit().beep();
        fireHyperlinkUpdate(new HyperlinkEvent
        ↪ (this, HyperlinkEvent.EventType.ACTIVATED,url));
        Utilities.getBasicService().showDocument(url);
    }
    public void mousePressed(MouseEvent e) {
    }
    public void mouseReleased(MouseEvent e) {
    }
    public void mouseDragged(MouseEvent e) {
    }
    public void mouseMoved(MouseEvent e) {
    }
    public void mouseEntered(MouseEvent e) {
        if (!alreadyClicked)
            super.setText("<html><font color="+COLOR_BEFORE_CLICKING+" >
        ↪<u>"+text+"</u></font>");
        else
            super.setText("<html><font color="+COLOR_AFTER_CLICKING+" >
        ↪<u>"+text+"</u></font>");
```

**LISTING 11.2** Continued

---

```
        setCursor(Cursor.getPredefinedCursor(Cursor.HAND_CURSOR));
    }
    public void mouseExited(MouseEvent e) {
        setText("<html><body>"+text+"");
    }
    public void setText(String txt) {
        text=txt;
        super.setText("<html><font color="+COLOR_BEFORE_CLICKING+" >
↳<u>"+text+"</u></font>");
    }
    public void setURL(URL u) {
        url=u;
        setToolTipText("view in browser "+url);
    }
    public void setURL(String u) {
        try {
            setURL(new URL(u));
        } catch (Exception exc) {
            System.out.println(getClass()+"setURL: "+ exc);
        }
    }
    public synchronized void removeHyperlinkListener(HyperlinkListener l) {
        if (hListeners != null && hListeners.contains(l)) {
            Vector v = (Vector) hListeners.clone();
            v.removeElement(l);
            hListeners = v;
        }
    }
    public synchronized void addHyperlinkListener(HyperlinkListener l) {
        Vector v;
        if (hListeners == null)
            v = new Vector(2);
        else v = (Vector) hListeners.clone();
        if (!v.contains(l)) {
            v.addElement(l);
            hListeners = v;
        }
    }
    protected void fireHyperlinkUpdate(HyperlinkEvent e) {
        if (hListeners != null) {
            Vector listeners = hListeners;
            int count = listeners.size();
            for (int i = 0; i < count; i++) {
                ((HyperlinkListener) listeners.elementAt(i)).hyperlinkUpdate;
```

**LISTING 11.2** Continued

```
}  
}  
}  
}
```

Note that the Hyperlink events code (methods `removeHyperlinkListener`, `addHyperlinkListener`, `fireHyperlinkUpdate`) is useful for listening to launch events from other classes. It could be safely dropped if we use our label only for launching a Web browser.

The most interesting line is probably line 28, in which the browser is pointed to the given URL using the `BasicService` JNLP class. This can be as simple as the following line of code:

```
Utilities.getBasicService().showDocument(url);
```

In order to use it in your code, just set the text to be displayed, like a normal `JLabel`, plus the URL to point to.

This little component could be very handy when used to create professional looking About dialogs, for instance.

## File Management

Handling files with the JNLP API requires the use of the following classes:

- `FileContents`. This class represents a file, both for reading and for writing, using the usual input and output streams. It is metered; that is, only a given amount of space (bytes) can be allocated for a file in order to allow JNLP Clients to maintain control on the local resources allocated to launched applications. Note that path information is absent. This means that applications obtaining this object (by means of one of the following three services: `FileOpenService`, `FileSaveService` and `PersistenceService`) can examine only the file contents.
- `FileOpenService`. Applications wanting to access files on the local system should use this service. For untrusted applications, it is the only way to access local files (which is, of course, subject to user authorization). This will pop up a File Open dialog box for choosing a file.
- `FileSaveService`. This class is similar to the `FileOpenService`, but this time the `JFileChooser` dialog box will save a file (represented as a `FileContents` object).
- `JNLPRandomAccessFile`. The JNLP API implementation of a random access file (a file treated as an array of bytes).

Any attempt to use other means than those four classes to access local files for untrusted applications is not permitted. Listing 11.3 shows an example of these classes used to obtain files and then process them.

**LISTING 11.3** The FileContentsExample Class

```
package com.marinilli.b2.c11;
import com.marinilli.b2.c11.util.*;
import javax.jnlp.*;
import java.io.*;
/**
 * Chapter 11
 * @author Mauro Marinilli
 * @version 1.0
 */
public class FileContentsExample {
    public FileContentsExample() {
        testRead();
        testWrite();
        writeAsRAF();
        System.exit(0);
    }
    private void testRead() {
        FileContents file = null;
        FileOpenService fs =
            (FileOpenService)Utilities.getService("javax.jnlp.FileOpenService");
        if (fs!=null) {
            try {
                file = fs.openFileDialog(null, null);
            } catch (Exception e) {
                System.out.println("openFile: "+e);
            }
        }
        if (file!=null) {
            process(file);
            readLines(file);
        }
        System.out.println("finished.");
    }
    private void testWrite() {
        System.out.println("testWrite()");
        FileContents file = null;
        FileSaveService fs =
            (FileSaveService)Utilities.getService("javax.jnlp.FileSaveService");
        if (fs!=null) {
```

**LISTING 11.3** Continued

```
    try {
        file =
            ((FileOpenService)Utilities.getService("javax.jnlp.FileOpenService"))
                .openFileDialog(null, null);
        FileContents outcome = fs.saveAsFileDialog(null,null,file);
        System.out.println("outcome "+outcome);
    } catch (Exception e) {
        System.out.println("openFile: "+e);
    }
}
if (file!=null) {
    writeData(file);
}
}
private void testWrite2() {
    FileContents file = null;
    FileSaveService fs =
        (FileSaveService)Utilities.getService("javax.jnlp.FileSaveService");
    if (fs!=null) {
        try {
            FileContents outcome = fs.saveAsFileDialog(null,null,file);
        } catch (Exception e) {
            System.out.println("openFile: "+e);
        }
    }
    if (file!=null) {
        writeData(file);
    }
}
public void readLines(FileContents file){
    try {
        InputStream input = file.getInputStream();
        BufferedReader reader = new BufferedReader(new InputStreamReader(input));
        String s = "";
        while ((s = reader.readLine()) != null) {
            System.out.println("\""+s+"\"");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
public byte[] readBytes(FileContents file){
    byte [] buffer = null;
    try {
```

**LISTING 11.3** Continued

```
        InputStream input = file.getInputStream();
        buffer = new byte[(int)file.getLength()];
        input.read(buffer);
        input.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return buffer;
}
public void process(FileContents file){
    try {
        InputStream input = file.getInputStream();
        int c;
        while ((c = input.read())!= -1) {
            System.out.write;
        }
        input.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
public void writeData(FileContents file){
    try {
        if (file.canWrite()) {
            DataOutputStream dos = new DataOutputStream
            ↪(file.getOutputStream(false));
            dos.writeInt(123);
            dos.writeFloat(123F);
            dos.close();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
public void writeAsRAF(){
    FileContents file = null;
    FileOpenService fs =
    (FileOpenService)Utilities.getService("javax.jnlp.FileOpenService");
    if (fs!=null) {
        try {
            file=fs.openFileDialog(null,null);
            if (file.canWrite()) {
                enlargeFile(file, 2028);
                if (file.getMaxLength() > file.getLength() ) {
```

**LISTING 11.3** Continued

```
        JNLPRandomAccessFile raf = file.getRandomAccessFile("rw");
        raf.seek(raf.length() - 1);
        raf.writeUTF("Last In file.");
        raf.close();
    }
} catch (Exception e) {
    e.printStackTrace();
}
}
}
private long enlargeFile(FileContents file, int extraLength){
    long length = 0L;
    if (file!=null)
        try {
            length = file.getLength();
            if (length + extraLength > file.getMaxLength()) {
                length = file.setMaxLength(length + 1024);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    return length;
}
public static void main(String[] args) {
    FileContentsExample fileContentsExample1 = new FileContentsExample();
}
}
```

Let's discuss some aspects of this code.

- The constructor (lines 12–15) calls some example methods that manipulate files on the local file system. Note that if your application is trusted (as described in Chapter 10, “Defining the Client Environment”), you do not need to pass through the JNLP services to access local files.
- The method `testRead` tries to open up a File dialog box for accessing a file on local disks. The method `process` (lines 93–99) does some simple processing on the opened file, by copying it four bytes at a time onto the standard output. The method `readLines` reads it as a text file, line by line.
- The method `testWrite` at lines 34–53—after having obtained a reference on a local file and having asked to save its own file on the file system—invokes the `writeData` method, beginning at line 105. This method, after checking the permissions on the given file, treats it as a data file, writing some numerical data into it.

- The method `writeAsRAF` treats the file like a random access file and, after having enlarged it, writes a string at the end of the file. The general-purpose `enlargeFile` method tries to enlarge specified files whenever allowed by the JNLP Client.
- Finally, the method `testWrite2` is similar to `testWrite`, but asks for a `saveAsFileDialog`. The method `readBytes` at line 81 returns the specified file as a big chunk of bytes in memory.

## Resource Management

The `DownloadService` and `DownloadServiceListener` classes are dedicated to the download and caching of resources, and are often used together with the `ExtensionInstallerService`.

The `DownloadService` can return information about cached parts (refer to Chapter 10), parts of extensions, or simple JAR files. This information can be returned only for items belonging to the application currently running. In other words, the information can be returned only for items that are mentioned in the main JNLP file or in referenced JNLP files.

A JAR file, a part, or an extension part could be loaded using the methods `loadResource`, `loadPart`, and `loadExtensionPart`. The `DownloadServiceListener` is used as an argument in these methods in order to track the download process.

When removing cached items with these methods, resources are rarely deleted immediately. Rather, JNLP Clients would regard them as no longer needed, and that they should be released when required in the future.

As an example of cache control, see Listing 11.4.

### LISTING 11.4 An Example Of Cache Control

---

```
package com.marinilli.b2.c11;
import com.marinilli.b2.c11.util.*;
import javax.jnlp.*;
import java.net.URL;
import java.io.*;
/**
 * Chapter 11 - Example of DownloadService
 * @author Mauro Marinilli
 * @version 1.0
 */
public class DownloadExample {
    public DownloadExample() {
        removeFromJNLPCache("anapp.jar");
    }
    public static void removeFromJNLPCache(String resourceName){
```

**LISTING 11.4** Continued

```
DownloadService ds = null;
ds = (DownloadService)Utilities.getService("javax.jnlp.DownloadService");
URL url = null;
try {
    url = new URL(Utilities.getBasicService().getCodeBase(), resourceName);
} catch(IOException exc) {
    System.out.println("Creating URL: "+exc);
}
try {
    ds.removeResource(url ,null);
    System.out.println("Resource "+url+" removed from JNLP cache.");
} catch(IOException exc) {
    System.out.println("removing resource from JNLP cache: "+exc);
}
}
```

**11**RUNTIME CLIENT  
SERVICES

## Persistence

Persistence services are achieved through the use of a `PersistenceService` class that saves `FileContents` objects given a URL as a key. The local file system works like a cache for this data that is always thought of as being a copy of data stored on the server. The whole concept is inspired by the cookies mechanism; the primary difference being that the space allowed on the disk is bigger and more flexible.

The URL-keys work hierarchically. That is, common directories are shared by applications sharing portions of their codebase.

For example, two applications having codebases can see each other application's data at locations `http://www.asite.org/pub/` and `http://www.asite.org/`:

- `http://www.asite.org/pub/app1/`
- `http://www.asite.org/pub/app2/`

Therefore, common data can be organized following remote paths on a Web server.

Each single entry (a pair of a URL as a key and a `FileContents` as a value) can be tagged regarding its relationship with the server-side data cache. Tags are represented as ints that can be accessed via the accessory methods `setTag()` and `getTag()`. We can have three possible states for data stored locally:

<i>Tag Value</i>	<i>Meaning</i>
dirty	Server doesn't have an up-to-date copy of the data
cached	Server has an up-to-date copy
temporary	Data that can always be recreated

These states are read by the JNLP Client when it needs to clear the cache. The deletion order is: temporary files first; then cached data; and, finally, items tagged as dirty.

## A Persistence Utility Class

Another utility class provided with this book, the `PersistenceStorage` class, is a wrapper of the `PersistenceService` object, so that you don't have to deal with `FileContents` or all the details typical of the new `jnlp` API. Instead, the interface of this utility class is very simple to use, and there is no additional knowledge of the `PersistenceService` API required in order to use it. Of course, if you need particular operations, you can always extract the wrapped `PersistenceService` object and manage it as required. However, this class will help in the majority of real cases.

It is useful for storing Java objects locally with a given key. It works much like a Hash table, saving objects and retrieving them by means of a key (a string or a full URL). The key string is used to create a URL based on the codebase value (see the `getUrl()` method at lines 102–111) in Listing 11.5.

To save a serializable object, just use the following code:

```
persistenceStorage1.write("key", serializableObject);
```

to read the following value:

```
Object serializableObject = persistenceStorage1.read("key");
```

Then, cast the obtained object as needed.

### LISTING 11.5 The PersistenceStorage Utility Class

```
package com.marinilli.b2.c11.util;
import com.marinilli.b2.c11.util.Utilities;
import javax.jnlp.*;
import java.net.URL;
import java.io.*;
import java.util.*;
/**
 * Chapter 11 - utility wrapper for PersistenceService
 * @author Mauro Marinilli
 * @version 1.0
```

**LISTING 11.5** Continued

```
*/
public class PersistentStorage {
    private PersistenceService persistenceService;
    private long DEFAULT_SIZE = 2048L;
    public PersistentStorage() {
        persistenceService =
(PersistenceService)Utilities.getService("javax.jnlp.PersistenceService");
    }
    public PersistentStorage(PersistenceService ps) {
        persistenceService = ps;
    }
    public void write(String keyString, Object value) {
        write(keyString, value, DEFAULT_SIZE);
    }
    public void write(String keyString, Object value, long maxLength) {
        write(getUrl(keyString), value, DEFAULT_SIZE);
    }
    public void write(URL url, Object value, long maxLength) {
        if (exists(url))
            removeEntry(url);
        try {
            persistenceService.create(url, maxLength);
            FileContents fc = persistenceService.get(url);
            ObjectOutputStream oos =
                new ObjectOutputStream(fc.getOutputStream(false));
            oos.writeObject(value);
            oos.close();
        } catch (IOException ioe) {
            System.out.println(getClass()+"write("+url+", "+value+"): "+ioe);
        }
    }
    public Object read(String keyString) {
        return read(getUrl(keyString));
    }
    public Object read(URL url) {
        if (!exists(url))
            return null;
        Object object = null;
        try {
            FileContents fc = persistenceService.get(url);
            ObjectInputStream ois = new ObjectInputStream(fc.getInputStream());
            object = ois.readObject();
        } catch (Exception e) {
```

**11**

**LISTING 11.5** Continued

---

```
        System.out.println(getClass()+" .read("+url+"): "+e);
    }
    return object;
}
public boolean exists(URL url){
    try {
        if (persistenceService.getNames(url).length>0)
            return true;
    } catch (Exception e) {
        System.out.println(getClass()+" .exists("+url+"): "+e);
    }
    return false;
}
public void synchronize(String keyString) {
    synchronize(getUrl(keyString));
}
public void synchronize(URL url) {
    try {
        if (persistenceService.getTag(url)==PersistenceService.DIRTY)
            persistenceService.setTag(url, persistenceService.CACHED);
    } catch (Exception e) {
        System.out.println(getClass()+" .synchronize("+url+"): "+e);
    }
}
public void removeEntry(URL url) {
    try {
        persistenceService.delete(url);
    } catch (Exception e) {
        System.out.println(getClass()+" .remove("+url+"): "+e);
    }
}
public List getEntries(String keyString) {
    return getEntries(getUrl(keyString));
}
public List getEntries(URL url) {
    List list =null;
    try {
        list=
            Collections.unmodifiableList(
                Arrays.asList(persistenceService.getNames(url)));
    } catch (Exception e) {
        System.out.println(getClass()+" .getEntries("+url+"): "+e);
    }
    return list;
}
```

**LISTING 11.5** Continued

```
}
public PersistenceService getPersistenceService() {
    return persistenceService;
}
public URL getUrl(String keyString) {
    URL codebase = Utilities.getBasicService().getCodeBase();
    URL url = null;
    try {
        url = new URL(codebase, keyString);
    } catch (IOException ioe) {
        System.out.println(getClass().getName() + ".getUrl(" + keyString + "): " + ioe);
    }
    return url;
}
}
```

This class provides the following wrapper methods:

- `public void write(String keyString, Object value)`. It persistently saves the given object at the given key location.
- `public void write(String keyString, Object value, long maxLength)`. The same as the previous method, but it also specifies the maximum space allocable to the object serialized on disk.
- `public void write(URL url, Object value, long maxLength)`. The same as the preceding method, but a URL is supplied to identify the resource. Note that if the URL is not based on the codebase value (that is, it is not rooted on the Web server that deployed the application), the service won't work.
- `public Object read(String keyString)` and `public Object read(URL url)`. Will return an object given a key, expressed as a string or a complete URL.
- `public boolean exists(URL url)`. Returns true if there are some data cached with that URL as a key.
- `public void synchronize(String keyString)` and `public void synchronize(URL url)`. Synchronize the data, tagging it as explained before.
- `public void removeEntry(URL url)`. Remove the given entry from the local cache.
- `public List getEntries(String keyString)`. Returns a List collection for all the entries stored at the given URL.

- `public PersistenceService getPersistenceService()`. Obtains the wrapped persistence service.
- `public URL getUrl(String keyString)`. It is the method used by the `PersistenceStorage` class to translate the key strings in URLs.

## Controlling Installations

The API for controlling the installation of resources (and applications) comprises the `DownloadService`, `DownloadServiceListener`, and `ExtensionInstallerService`. We have already seen many examples of these classes at work. In Chapter 13, “A Complete Example,” there are also some examples of these classes in use.

In Listing 11.6, we present an interesting example of an installer that puts downloaded JAR files into a specified directory—here the `lib/ext/` directory of the currently installed JRE.

### LISTING 11.6 An Optional Packages Installer Example

---

```
package com.marinilli.b2.c11;
import com.marinilli.b2.c11.util.Utilities;
import javax.swing.*;
import java.awt.print.*;
import javax.jnlp.*;
import java.awt.*;
import java.io.*;
import java.net.*;
/**
 * Chapter 11 - Another Installer Example
 * @author Mauro Marinilli
 * @version 1.0
 */
public class InstallerExample implements Runnable {
    ExtensionInstallerService eis;
    public InstallerExample() {
        Utilities.getInstance("installer-msgs");
        eis = (ExtensionInstallerService)Utilities.getService(
        ↪ "javax.jnlp.ExtensionInstallerService");
        Thread t = new Thread(this);
        t.start();
    }
    public void run() {
        setupDownloadWindow();
        doResourcesDownload();
        installationCompleted(true, false);
    }
}
```

**LISTING 11.6** Continued

```
public void setupDownloadWindow() {
    eis.setHeading(Utilities.getMsg("heading1"));
    eis.setStatus(Utilities.getMsg("sometext"));
}
public void doResourcesDownload(){
    installOptionalPackage();
}
public void installationCompleted(boolean success, boolean reboot){
    if (success){
        if (reboot){
            //handle some reboot preparation
        }
        eis.installSucceeded(reboot);
    } else {
        //handle failure
        eis.installFailed();
    }
}
public void installOptionalPackage(){
    String optionalPackagesJARDirWin = "\\lib\\ext\\"; //see Sun docs
    String jarName = "anapp.jar";
    String jarServerPath = "http://server/b2/c11/"+jarName;
    URL jreURL = null;
    try {
        jreURL = new
URL("http://jsp.java.sun.com/servlet/javawsExtensionServer");
    } catch (Exception e) {
        System.out.println("creating URL: "+e);
    }
    String jrePath = eis.getInstalledJRE(jreURL, "1.3.0");
    File javaFile = new File(jrePath); //where java is located
    String jreRoot = javaFile.getParentFile().getParent();
    File libDir = new File(jreRoot+optionalPackagesJARDirWin);
    if (!libDir.exists())
        libDir.mkdirs();
    File jarFile = new File(jreRoot + optionalPackagesJARDirWin +jarName);
    try {
        FileOutputStream fos = new FileOutputStream(jarFile);
        URL jarURL = new URL(jarServerPath);
        URLConnection jarConn = jarURL.openConnection();
        InputStream is = jarConn.getInputStream();
        int q;
        while ((q = is.read()) != -1)
            fos.write(q);
    }
```

**11**

**LISTING 11.6** Continued

---

```
        is.close();
        fos.close();
    } catch (Exception e) {
        System.out.println("creating & writing "+jarName+": "+e);
    }
}
public static void main(String[] args) {
    InstallerExample installer = new InstallerExample();
}
}
```

---

There are some interesting things to notice in this listing:

- First, notice the *sequence* of actions that an installer usually follows. First, it obtains the `ExtensionInstallerService` (line 17–18); then, it prepares the GUI, both creating it by itself or adapting the JNLP Client’s one, using methods of the previously obtained `ExtensionInstallerService` (method `setupDownloadWindow` at lines 27–30). Then, it performs its operations in the method `doResourcesDownload` (lines 31–33), eventually calling `setJREInfo` or `setNativeLibraryInfo` for notifying changes to the JRE, and finally concludes the installation procedure, as in method `installationCompleted` at lines 34–44.
- Also, notice how to locate the directory where a JRE is currently installed, by using the following, as shown in Line 55:  

```
extensionInstallerService.getInstalledJRE(jreURL,VersionString);
```

Note that the JRE URL is the vendor’s URL, and it can be determined together with the exact version string directly from the Preferences control panel of Java Web Start or in an analogous way for other JNLP Clients.
- The method `installOptionalPackage` beginning at line 45 first tries to determine where the JRE Java executable is located. Then, it creates the proper path where the resources are intended to be downloaded.
- Lastly, just a note on threads. Despite the fact that they are not needed in this simple example, they are commonly used by installers that want to reduce the total installation time.

The JNLP main file (namely `installer-main.jnlp`) shown in Listing 11.7 declares its own application JAR files and the installer extension, as reported in Listing 11.8.

**LISTING 11.7** The Main JNLP file for the Optional Packages Installer Example

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+"
  codebase="http://server/b2/c11/">
  <information>
    <title>Optional Pkg. Install.</title>
    <vendor>Mauro's Workshop</vendor>
    <offline/>
  </information>
  <resources>
    <j2se version="1.2+" />
    <extension
      name="Installer"
      href="installer-ext.jnlp">
    </extension>
    <jar href="dummy-app.jar" />
  </resources>
  <application-desc main-class="com.marinilli.b2.c11.DummyApp" />
</jnlp>
```

The JNLP extension descriptor file is listed below in Listing 11.8. Note that to access the file system, we require signed installers specifying trusted permissions (that is, `all-permissions` or `j2see-application-client-permissions` values) and that optional packages are located in different directories within a JRE environment, depending on the platform. Consequently, the platform-specific code has to be specified in the `resources` element, or the Java code should be able to tell the difference. See the J2SE documentation on optional packages for more details. For simplicity, we assumed a JRE exists that is compliant with the Windows environment.

**LISTING 11.8** The Extension Descriptor JNLP file for the Optional Packages Installer Example

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+"
  codebase="http://server/b2/c11/">
  <information>
    <title>Installer Example</title>
    <vendor>Mauro's Workshop</vendor>
    <offline/>
  </information>
  <security><all-permissions/></security>
  <resources>
    <j2se version="1.2+" />
```

**LISTING 11.8** Continued

```
<jar href="installer-signed.jar" />
</resources>
<installer-desc main-class="com.marinilli.b2.c11.InstallerExample" />
</jnlp>
```

**NOTE**

Using the JNLP API, installers may even ask a JNLP Client to reboot the computer to finish the installation. Refer to line 39 in Listing 11.6.

In order to install a new JRE, the procedure is similar to the `InstallerExample` previously discussed. However, the method `setJREInfo` must be called by the installer, providing the absolute path to the java executable of the installed JRE. Likewise, when installing a native library in a particular directory in the local file system, it is necessary to invoke the `setNativeLibraryInfo` to inform the JNLP Client from where to download the native libraries.

## Other Services

JNLP also offers implementations of printing and Clipboard access services, much as in the case of file-related services. This allows untrusted applications to have a chance to access the given services.

### Accessing the System Clipboard

Untrusted applications cannot access the system Clipboard without passing through this service. See the `AnApplication` class (Listing 11.10) for an example of usage of this service. In particular, examine lines 75–86 of Listing 11.10.

### Queuing Jobs to the Printer

For printing applications via the JNLP API (for unsigned applications as well) use the `PrintService`. See Listing 11.9 for an example usage of this service.

**LISTING 11.9** An Example Of Printing with the JNLP API

```
package com.marinilli.b2.c11;
import com.marinilli.b2.c11.util.Utilities;
import java.awt.print.*;
```

**LISTING 11.9** Continued

```
import javax.jnlp.*;
import java.awt.*;
/**
 * Chapter 11 - An example of printing via JNLP API
 * @author Mauro Marinilli
 * @version 1.0
 */
public class PrintServiceExample {
    public PrintServiceExample() {
        PrintService ps =
            (PrintService)Utilities.getService("javax.jnlp.PrintService");
        ps.print(new PrintableExample());
        System.exit(0);
    }
    public class PrintableExample implements Printable {
        public int print(Graphics graphics, PageFormat pageFormat, int pageIndex){
            Graphics2D g2d = (Graphics2D)graphics;
            g2d.setPaint(Color.black);
            g2d.drawString("A JNLP Print",100,100);
            return PAGE_EXISTS;
        }
    }
    public static void main(String[] args) {
        PrintServiceExample printServiceExample1 = new PrintServiceExample();
    }
}
```

**11**RUNTIME CLIENT  
SERVICES

## An Example Application

As a recapping example of what said, let's take a look at Listing 11.10, in which an application uses many of the things discussed in this chapter. As we can see, we are constantly moving toward a full-fledged application, such as the one presented in Chapter 13.

**LISTING 11.10** A Recapping Example of JNLP Runtime Services at Work

```
package com.marinilli.b2.c11;
import com.marinilli.b2.c11.util.*;
import javax.swing.*;
import java.awt.event.*;
import javax.jnlp.*;
import java.net.URL;
import java.io.*;
import java.util.Iterator;
```

**LISTING 11.10** Continued

```
import java.awt.*;
/**
 * Chapter 11 A test application
 * @author Mauro Marinilli
 * @version 1.0
 */
public class AnApplication extends JFrame {
    JPanel centerPanel = new JPanel();
    JButton pasteButton = new JButton();
    JLabel southLabel = new JLabel();
    JLabel centerLabel = new JLabel();
    JPanel northPanel = new JPanel();
    JNLPLabel northLabel = new JNLPLabel();
    JButton openFileButton = new JButton();
    public AnApplication() {
        setTitle(Utilities.getMsg("title"));
        setIconImage(Utilities.getImageIcon("bubu").getImage());
        this.getContentPane().setLayout(new BorderLayout());
        this.addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        centerPanel.setLayout(new BorderLayout());
        pasteButton.setText("Paste");
        pasteButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                centerLabel.setText(pasteClipboardContent());
            }
        });
        southLabel.setText(Utilities.getMsg("hello"));
        centerLabel.setIcon(Utilities.getImageIcon("anIcon.gif"));
        northLabel.setText("please click here!");
        northLabel.setURL("http://server/b2/index.html");
        openFileButton.setIcon(Utilities.getImageIcon("open.gif"));
        openFileButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(ActionEvent e) {
                openFile();
            }
        });
        this.getContentPane().add(centerPanel, BorderLayout.CENTER);
        centerPanel.add(pasteButton, BorderLayout.EAST);
        centerPanel.add(centerLabel, BorderLayout.CENTER);
        centerPanel.add(northPanel, BorderLayout.NORTH);
    }
}
```

**LISTING 11.10** Continued

```
northPanel.add(openFileButton, null);
northPanel.add(northLabel, null);
this.getContentPane().add(southLabel, BorderLayout.SOUTH);
PersistenceService ps =

(PersistenceService)Utilities.getService("javax.jnlp.PersistenceService");
if (ps!=null) {
    doSomething(ps);
}
setSize(300,200);
setVisible(true);
}
private void doSomething(PersistenceService ps){
    PersistentStorage pst = new PersistentStorage();
    pst.write("url0","salve");
    pst.synchronize("url0");

    Iterator iter = pst.getEntries("url0").iterator();
    while (iter.hasNext()) {
        Object obj = iter.next();
        System.out.println("item="+obj);
    }
}
private String pasteClipboardContent() {
    String s = "";
    ClipboardService cbs =
        (ClipboardService)Utilities.getService("javax.jnlp.ClipboardService");
    try {
        if (cbs.getContents()!=null)

s=(String)cbs.getContents().getTransferData(java.awt.datatransfer.DataFlavor.st
ringFlavor);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return s;
    }
    public void openFile() {
        String[] suffixes = {"txt","jnlp","jar","java"};
        FileContents file = null;
        FileOpenService fs =
            (FileOpenService)Utilities.getService("javax.jnlp.FileOpenService");
        if (fs!=null) {
            try {
```

**LISTING 11.10** Continued

```
        file = fs.openFileDialog("mydir", suffixes);
    } catch (Exception e) {
        System.out.println("openFile: "+e);
    }
}
}
public static void main(String[] args) {
    AnApplication anApplication1 = new AnApplication();
}
}
```

Figure 11.2 presents a screenshot from the test application running.

Production direc-  
tive said to crop  
this. Okay?



**FIGURE 11.2**

*A screenshot from the test application.*

In Figure 11.2, we see several different things:

- Locale-specific data, both in the title and in the bottom label. If you try to run the program on a computer with a `Locale` other than Italian, you'll see English text instead (provided as default; see following).
- Icons loaded and icons not found in the JAR file. The `JFrame` icon hasn't been found so the `Utilities` class provides a default one without firing any exception.
- The `JNLPLabel` at work, showing the pointed URL as a tooltip (the hand cursor is not shown).
- Clicking the `Paste` button will activate the `Clipboard`, which will prompt for authorization because the application is not signed. The button with the open file icon, instead, will prompt to open a file from the local file system.

Note that, besides the launching JNLP file (`anapp.jnlp`), there is the `anapp.jar` JAR file containing the following items:

- Two properties files, `messages.properties` and `messages_it.properties`, specifying text for the two supported locales.
- Java classes needed to run the application.
- Eventually, the `META-INF/MANIFEST.MF`, necessary if the main class has not been specified in the JNLP file.
- Some graphic icons.

See Chapter 13 for more examples of use of the JNLP API.

## Summary

In this chapter, we examined the runtime services offered by JNLP client to its applications. The heart of these services lies within the `javax.jnlp.ServiceManager` class. In Chapter 12, we will study how JNLP is supported by a remote server which acts as a deployment server, as described in Chapter 2.

