10 The Preferences API

This chapter covers

- Purpose and scope of the API
- Comparison with other persistence mechanisms
- Reading and writing data
- Change listeners

The Preferences API, in the `java.util.prefs` package, gives the programmer a convenient way to save and recall configuration information. This information can be saved during one invocation of a program and recalled during a later invocation—that is, the information is *persistent* across invocations. Additionally, this information can be shared between invocations, if desired.

In a sense, the Preferences API could be considered to be misnamed, since it can be used to store any kind of information, not just preference data. However, because of size limitations, and because the Preferences API makes no claims of efficiency, it should not be used as a general database. It is best used for configuration information—small pieces of information that are read and written one or a few times per program invocation, and which need to be saved between invocations. Generally, data stored in the Preferences API consists of a few values that describe the configuration of the program as customized explicitly or implicitly by the user, or the configuration of the software installation itself.

10.1 What the Preferences API is for

Looked at abstractly, the Preferences API is just another way to store data. Java already has a number of ways of doing this, and more are being added all the time. It's important to be clear about what the Preferences API is good for, and what it is not good for.

10.1.1 Simple Preferences API example

Suppose you've written a program that has a number of independent windows, and you want to save their locations on screen as a convenience to the user. That way, once he has arranged the windows the way he likes them, they will be displayed that way the next time he runs the program.

Here is how this can be done with the Preferences API:

```
import java.util.prefs.*;

// ...

Preferences prefs = Preferences.userNodeForPackage( getClass() );
prefs.putInt( "main window x", mainWindowX );
prefs.putInt( "main window y", mainWindowY );
prefs.putInt( "listing window x", listingWindowX );
prefs.putInt( "listing window y", listingWindowY );
```

And here is how these values can be retrieved during a later invocation:

```
Preferences prefs = Preferences.userNodeForPackage( getClass() );
mainWindowX = prefs.getInt( "main window x", 200 );
mainWindowY = prefs.getInt( "main window y", 200 );
```

```
listingWindowX = prefs.getInt( "listing window x", 400 );
listingWindowY = prefs.getInt( "listing window y", 400 );
```

Note that the names of the preference values can have spaces—this is a signal that these names should be thought of as human-readable, and even user-friendly. The Preferences API is to be used for this kind of data.

10.1.2 Appropriate applications of the Preferences API

As mentioned previously, the Preferences API can store just about any kind of data. However, it isn't intended for all possible data storage purposes. Data that is being managed using the Preferences API should meet the following criteria:

- The data should be small.
- The data should apply to a single user, or to a single installation.
- The data should need to be stored between invocations.
- The data should only be accessed occasionally.
- The data should not be critical.
- The data should not be essential to the functioning of the application.

Data that meets these criteria is often called *preference* data. This is in contrast to *application* data, which is defined as data that is central to the operation of an application, or that is created or generated by the user in the normal operation of the application.

These criteria are important enough that some elaboration is in order:

The data should be small

This is an intentionally vague statement. The Preferences API specifies a maximum size of 8,192 bytes per value, and so there is a hard limit to what can be stored in a single slot. In addition to this, good judgment should be used. For example, in a word processing program, the Preferences API would be used to store the user's preferred font, but it would not be used to store the contents of a document that is created.

The amount of data should not vary widely during the use of the program. In theory, you *could*, for example, use the Preferences API to store telephone numbers in a Rolodex-style application. But this data could consist of 20 phone numbers, or it could consist of 20 million phone numbers. No existing implementation of this API is capable of dealing efficiently with that much data.

The data should apply to a single user, or to a single installation

The Preferences API is intended to store information that pertains to a particular user, such as changes a user has made to the configuration settings of the program. Thus, in a multiwindow chat program, the positions of the windows might be stored using the Preferences API, but global server log data would be stored in the filesystem.

The data should need to be stored between invocations

This is, in a sense, the whole point. While the API could be used to store data during a single invocation of the program, this would be overkill, since such data could be stored more easily in variables.

It is very common to reach for the Preferences API precisely because you've realized that a particular piece of data, which previously was a regular program variable, needs to be remembered between invocations.

As an example, a game server would not use the Preferences API to store the current scores of players as they are playing the game, but it might store the high-score lists.

The data should only be accessed occasionally

This is another intentionally vague statement. There is, of course, no limit to how often you can access data stored via the Preferences API. However, the API is not intended primarily to be fast. It is only intended to store its data reliably.

It would be common for an application to read configuration data via the Preferences API during startup and never use the API again. Just as commonly, a program might write out changes to configuration values either at shutdown or at the moment the user requests the changes. Some applications, such as servers, might be instructed to reread configuration data from time to time, allowing the applications to be reconfigured without having to shut down and start up again. In other applications, configuration data might be read or written once in a while, such as every time a user connects to the system.

The Preferences API is not intended to store data that is updated thousands of times a second. While a fast enough computer could possibly handle it, such data is really best served by a dedicated database engine or a very fast filesystem.

As an example, a sophisticated web-server logging module might use the Preferences API to store the configuration determining which data should be logged, but it should not use the API to store the actual log data.

The data should not be critical

Applications that store data in the filesystem, or in a database, should continue to do so. These are the places where users and system administrators expect to find data, and the Preferences API is not meant to change that. Corrupted Preferences data should be considered a loss of configuration data, not of application data.

As an example, a program that is used to perform lengthy mathematical calculations might use the Preferences API to store the user's choice of which of several calculation methods to use, but the results of the calculations should be stored in the filesystem, or in a database.

The data should not be essential to the functioning of the application

It is an explicit design goal of the Preferences API that it should not require access to the underlying data store for it to function properly. Configuration information is not considered essential to the functioning of an application. Often, it consists of settings that are entered manually by the user and then remembered for convenience. Such information can be entered again, if necessary. Thus, any data that is central to the execution of the program should not be stored in the Preferences API.

As an example, you might use the Preferences API to store a user's address and phone number if the application were a word processor or spreadsheet, but you would not use it to store such information if the application were an address-and-telephone database.

10.1.3 Design goals of the Preferences API

The following is a list of goals that inspired the design of the Preferences API. These goals distinguish this API from other persistence mechanisms:

- The API should provide a hierarchical, tree-like data store.
- The API should store primitive data types.
- The API should guarantee back-end neutrality.
- There should be no need to remember locations of files.
- The API should not require explicit saving and loading of data.
- The API should be permitted to operate asynchronously.
- Data from different packages or applications should not interfere with each other.
- The API should work in a multithreaded environment.
- The API should work in a multiprocess environment.
- The API should work in a multilanguage environment.

- The API should provide only the minimum concurrency protection.
- The API should work even if the backing store is not available.
- The API should supply per-user data and system-wide data.

We'll take a closer look at each of these goals, because they go a long way toward clarifying when and how this API should be used:

The API should provide a hierarchical, tree-like data store

As we see in section 10.3, the data model used by the Preferences API is *hierarchical*, as opposed to relational or flat. This provides the programmer with the best trade-off between simplicity and flexibility.

The API should store primitive data types

The API provides direct support for storing strings, numerical types (integers, floats, doubles), booleans, and small byte-arrays. There is no support for serialized objects.

The API should guarantee back-end neutrality

It is an explicit and critical feature of the Preferences API that it can and should be implemented differently on different systems. The implementation itself is divided into a system-independent portion that is identical on all systems, and a system-dependent back end.

Unlike most database engines, the Preferences API should behave identically on all systems. This means that any differences in the data-storage capabilities must be hidden by the system-dependent layer. There are no optional methods (but see the discussion of stored defaults in 10.6). Using the API's import/export facility, it should be possible to export a preferences database from one system and import it cleanly into another system.

The repository where the data is actually stored is called the *backing store*. This is distinguished from the Preferences API, or *front end*, which enforces the structures and procedures in a platform-independent way. The backing store might have completely different semantics from the Preferences API, in which case it is the responsibility of the implementation to build the desired semantics on top of the semantics of the back end.

There should be no need to remember locations of files

One of the disadvantages of the older `java.util.Properties` method of storing configuration data is that the programmer was required to find and load the properties files. This made it harder to share configuration data between applications, or to establish conventions for configuration data, because there was no specification telling the applications where the properties files were located.

The Preferences API deals with this by creating a global database whose location is irrelevant to the programmer.

The API should not require explicit saving and loading of data

Using `java.util.Properties`, the programmer would use get and set methods to access data values, and load and store methods to commit these values to disk. In the Preferences API, only the get and set methods are needed—loading and storing are taken care of automatically (although the API provides `flush()` and `sync()` methods—see sections 10.4.15 and 10.4.16).

The API should be permitted to operate asynchronously

The implementation is allowed to defer the writing of data values. The actual API writing calls can return immediately without the actual data being written to the backing store. This allows for backing stores that are slow or only intermittently available.

Data from different packages or applications should not interfere with each other

The user is encouraged by the API to store configuration data separately for each package. However, this is not enforced, which means that it is also easy to share data between packages or applications.

The API should work in a multithreaded environment

The API should be thread-safe.

The API should work in a multiprocess environment

The API should work even if multiple instances of the Preferences API, within multiple JVMs, are accessing the same backing store.

The API should work in a multilanguage environment

The API should work even if programs written in other languages, using other libraries, are accessing the same backing store.

The API should provide only the minimum concurrency protection

Despite the fact that the Preferences API must support the existence of multiple writers to the same backing store, it is not intended to supply sophisticated concurrency mechanisms. There is no support for transactions of any kind. Atomicity is at the level of the single key/value pair. There is no way to ensure that a collection of multiple changes are committed either all together, or not at all.

Within a multithreaded Java program, a sequence of accesses in multiple threads is defined as having the same semantics as they would if they were carried out in a single thread—although the order of these accesses is not specified. This is really just a fancy way of saying that no single access will be interrupted by any other single access, even if the accesses are coming from threads that are active at the same time.

The API should work even if the backing store is not available

It is commonly understood that an application should continue to work even if its configuration files are deleted. In such a situation, the application should run as best it can using a default configuration. The Preferences API is designed to reflect this.

The default implementations that come with the standard JDK packages will likely be such that configuration data is always available to applications. However, it is quite conceivable that the implementation for a small wireless device might store configuration data on a central server. In the event that the device cannot reach the central server, its applications should still be able to run.

The API should supply per-user data and system-wide data

The Preferences API recognizes that some configuration data is configuration for the *system*, and some is for the *user*. Following this distinction, the preferences database consists of a section for system data, and a section for user data. There is a separate user database for each user, and the selection between these is handled automatically and invisibly by the API.

10.2 Knowing when to use the Preferences API

As mentioned earlier, there are many ways to save data in Java. The Preferences API isn't even the only way to save preference-style data. However, for most data of this kind, the Preferences API is the new *definitive* method for storing such data. In this section, we look at some competing methods and compare them with the Preferences API.

10.2.1 Comparison with java.util.Properties

The Preferences API package is meant to replace "most common uses of Properties," according to the *Preferences API Design FAQ*. In general, you should think of the `java.util.prefs` package as the new version of the `java.util.Properties` class.

The main deficiencies of the `Properties` class, in relation to the Preferences API, are as follows:

- `Properties` only deals with string data.

- `Properties` requires the programmer to load and save the data to and from a file or stream.

10.2.2 Comparison with JNDI

A thorough comparison between the Java Naming and Directory Interface (JNDI) and the Preferences API package is beyond the scope of this book. Suffice it to say that while JNDI can certainly help you accomplish the same goals as the Preferences API, it does so with the conceptual and computational overhead of a sophisticated system consisting of many classes spread through a number of packages. The Preferences API is intended to be available on any platform, no matter how small, and is meant to be easy to use. It is also meant to be used in situations where it's okay if the data isn't available—which may not be the case for your application.

10.3 Understanding the data hierarchy

The Preferences API provides a particular *model* for the data it stores. The model is loosely based on a filesystem, but has *names* and *values* rather than *filenames* and *file contents*. The API provides two separate *trees* of data—a system tree and a user tree.

10.3.1 Tree-like structure

The Preferences API provides a *tree-like* data model. This is in contrast to other common database structures, such as *relational* and *flat-file*. The database itself is structured as a set of nodes, and each node can contain a set of *key/value pairs*. A node can contain other nodes—a node that contains another node is called the *parent* of that node, and the contained node is called the *child* of the parent node.

As we'll see in section 10.3.3, there are two distinct trees—a *user* tree and a *system* tree.

10.3.2 Key/value pairs

Each key/value pair is an *association* between the name (the key) and the value. You can store a value under a name, and then use that name later to retrieve the stored value.

A name must meet the following requirements:

- It must be a primitive Java string
- It cannot be the null string
- It cannot contain the character "/"

A value must be one of the following primitive Java types:

- string (`String`)
- boolean (`boolean`)
- integer (`int`)
- long integer (`long`)
- floating-point (`float`)
- double-precision floating-point (`double`)
- byte array (`byte[]`)

Furthermore, any value that has the potential to be large, such as a string or byte array, must be less than `Preferences.MAX_VALUE_LENGTH` (8,192) bytes in size.

10.3.3 System vs. user

As was mentioned previously, the preferences database is divided into two main sections. The *user* tree is used to store configuration data on a per-user basis, while the *system* tree is used to store configuration data on a per-system (or per-installation) basis.

What this means in practice is that a program has access during its execution to two distinct data trees. These trees are functionally identical, with identical APIs. They differ in the kind of data that is stored in them, and in the way they are made available to a particular user.

There is a user tree for every single user on the system. A running program only has access to one of these user trees at a time—the user tree corresponding to the current user. At the same time, the program has access to the system tree. There is only one system tree, and it is shared among all running applications in the system.

It is important to note that the system tree is in no way protected from modification. The user/system distinction does not provide any form of security.

10.3.4 Definition of a user

The Preferences API does not specify any kind of formal definition of what a *user* is, nor does it involve itself with any kind of user authentication. It does not even require that a user have a name. All that is required is that the `userNodeForPackage()` and `userRoot()` methods be defined.

It is assumed that "user" will be defined in a system-dependent way, and that there is a way of determining the user at runtime. There are no security protections for this determination except as supplied by the underlying implementation.

10.3.5 Pathnames

As mentioned previously, both the user and system halves of the preferences database are structured like trees, with nodes that can contain key/value pairs and other nodes. This structure is designed to resemble that of a filesystem, and like a filesystem, each element in a tree has an *absolute pathname*, which specifies its precise location within the tree. A node or key/value pair can also have a *relative pathname*, which specifies its location in relation to another node within the same tree.

Figure 10.1 shows an example configuration of nodes and key/value pairs. These nodes and values have the absolute pathnames listed in table 10.1.

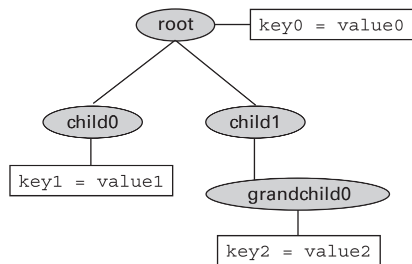

Figure 10.1 An example configuration of nodes and key/value pairs. Nodes are shown as ovals, while key/value pairs are shown as rectangles.

Relative to child1, the nodes and values have the relative pathnames listed in table 10.2.

Table 10.1 Each node in figure 10.1 has an absolute pathname. This pathname describes the location of the node in relation to the root of the preferences tree.

Node	Absolute pathname
root	/
child0	/child0
child1	/child1
grandchild0	/child1/grandchild0

Table 10.2 Nodes also have relative pathnames, which show their relationship to other nodes, rather than to the root of the tree.

Node	Relative pathname
`child1`	`""`
`grandchild0`	`grandchild0`

Note that there is no equivalent to the "`..`" found in some operating systems. All relative pathnames go toward the leaves of the tree.

10.3.6 Per-package subtrees

The Preferences API encourages the programmer to subdivide the preference trees by package. This helps keep the data for different programs from interfering with each other.

Each package name can be turned into an absolute pathname by changing each "`.`" to a "`/`" and adding a "`/`" at the front of the string. For example, `java.awt.font` becomes `/java/awt/font`. Code within the `java.awt.font` package would then store its subnodes and key/value pairs underneath this node. This is described in more detail in the next section.

Note that this naming convention is just that—a convention. However, as we'll see, this convention is employed by the Preferences API itself, in methods like `userNodeForPackage()` and `systemNodeForPackage()`, so you should stick to it unless you have a good reason not to.

10.4 Using the API

In this section, we'll take a look at the basic usage of the Preferences API. For the most part, the examples in the following sections use the user preferences tree. Unless otherwise noted, the examples work identically when used with nodes from the system tree.

10.4.1 Traversing the data hierarchy

Before you can read or write any data using the Preferences API, you must get your hands on a `Preferences` object. This object corresponds to a single node in a preferences tree, and it gives you direct access to the key/value pairs within that node. It also allows you to acquire `Preferences` objects for other nodes, particularly the node's children.

Once you have access to a preferences node, you can read and write values, and use it to get access to children, grandchildren, and so on. Remember that the Preferences API data hierarchy is a lot like a filesystem. When using a filesystem, your application is free to create subdirectories, subsubdirectories, and so on, and to store any data in these directories, using any filenames. Analogously, with the Preferences API, your application is free to create child nodes, grandchild nodes, and so on, and to store any values in these nodes, using any keys. Remember, however, that the Preferences API provides *two* separate data hierarchies—a user hierarchy and a system hierarchy. The former is used for user-specific data, while the latter is used for non-user-specific (or system- or application-wide) data. In this section, we'll take a look at the methods used to get access to different nodes in the data hierarchy.

There are a number of ways of getting a `Preferences` object. The simplest way is to use the `userRoot()` method, which provides access to the root of the user preferences tree. We'll look at other methods for getting access to a node later in this section. (All of these methods have analogous `system` methods.)

```
Preferences root = Preferences.userRoot();
```

This `Preferences` object corresponds to the path `"/"`.

Once you have your hands on the `Preferences` object for a particular node, you can use it to gain access to descendants of that node. This is done using the `node()` method, which takes a relative or absolute pathname and returns a `Preferences` object.

```
Preferences child0 = root.node( "child0" );
```

Since `root` corresponds to the path "`/`", `child0` corresponds to the path "`/child0`". The following line has the same result as the preceding line:

```
Preferences child0 = root.node( "/child0" );
```

You can call the `node()` method from *any* `Preferences` object, not just `root`. By doing this you can use any `Preferences` object to get access to one of its children:

```
Preferences child1 = root.node( "child1" );
Preferences grandchild0 = child1.node( "grandchild0" );
```

The object `grandchild0` corresponds to the absolute path "/child1/grandchild0".

Note that when you use an absolute path, the path of the old object does not matter. The following four lines are equivalent:

```
Preferences grandchild0 = child1.node( "grandchild0" );

Preferences grandchild0 = child1.node( "/child1/grandchild0" );

Preferences grandchild0 = root.node( "/child1/grandchild0" );

Preferences grandchild0 = grandchild0.node( "/child1/grandchild0" );
```

Each package is given its own subtree within the data hierarchy. The most common place for Java code to store its data is in the subtree corresponding to its package, and you can get access to a package's node using the `userNodeForPackage()` method, which is a static method of `Preferences`:

```
Preferences prefs = Preferences.userNodeForPackage( getClass() );
```

Note that you must supply a class to this method. `userNodeForPackage()` gets the package name of the supplied class and turns this into an absolute pathname. This pathname is then used to get access to the preferences node. Thus, the following two code fragments are equivalent:

```
Preferences prefs = Preferences.userRoot().node( "/java/util/prefs" );
```

```
package java.util.prefs;
//...
Preferences prefs = Preferences.userNodeForPackage( getClass() );
```

In the second of the preceding code fragments, we use the `getClass()` method (shown in boldface) to get the `Class` object corresponding to the class containing this code. This is passed to the `userNodeForPackage()` method in order to get the Preferences node corresponding to the package containing this class.

However, you can't use the `getClass()` method if you're writing a static method; in this case, you can use the `<Classname>.class` syntax to get the `Class` object associated with a static method:

```
public class MyClass
{
    static private void method() {
      Preferences prefs = Preferences.userNodeForPackage( MyClass.class );
      System.out.println( prefs );
    }
}
```

Just as you can get the user node for a particular package, you can get the system node for that package using `systemNodeForPackage()`:

```
Preferences prefs = Preferences.systemNodeForPackage( getClass() );
```

As mentioned earlier, the system node for a package can be used in the same way as the user node, but it is meant for data that is not user-specific.

In Java, code that is not assigned to a particular package is placed in the *default package*. Within the Preferences API, the absolute pathname for this package is "`/<unnamed>`".

10.4.2 Reading and writing values

Once you have a `Preferences` object, you can use it to read and write values. The following fragment writes an integer to the `Preferences` node for the current class package:

```
int windowX = 250;
Preferences prefs = Preferences.userNodeForPackage( this );
prefs.putInt( "window x", windowX );
```

Getting the value back out later is done as follows:

```
Preferences prefs = Preferences.userNodeForPackage( this );
int windowX = prefs.getInt( "window x", 300 );
```

Note that `getInt()` takes a second argument—this is a *default* value, which will be returned from `getInt()` if there isn't a value for `window x`. Default values are *always* used in any `get()` call. (See section 10.4.7 for more information.)

The same thing can be done with other types:

```
boolean showGrid = false;
Preferences prefs = Preferences.userNodeForPackage( this );
prefs.putBoolean( "show grid", showGrid );
```

```
Preferences prefs = Preferences.userNodeForPackage( this );
float gravity = prefs.getFloat( "gravity", 9.8 );
```

There is a `get()` method for each basic type. Note that the values are always stored as strings and that type checking is not performed. Values that are the wrong type, and that therefore cannot be parsed, will trigger the appropriate format exception.

10.4.3 Allowable types

Table 10.3 shows the types that are directly supported by the Preferences API, along with the methods that read and write them. Note that all of the types are primitive Java types except for strings (which are objects) and byte arrays (which are arrays).

Table 10.3 The Preferences API has `get()` and `put()` methods for each of the basic types, as well as for string and byte array types.

Type	Java name	Reader method	Writer method
string	`String`	`get()`	`put()`
byte array	`byte[]`	`getByteArray()`	`putByteArray()`
boolean	`boolean`	`getBoolean()`	`putBoolean()`
floating-point	`float`	`getFloat()`	`putFloat()`
double-precision floating-point	`double`	`getDouble()`	`putDouble()`
integer	`int`	`getInt()`	`putInt()`
long integer	`long`	`getLong()`	`putLong()`

10.4.4 Allowable keys

You aren't required to take your key string from the name of the variables being written. The following lines of code are all valid:

```
int windowX = 250;
Preferences prefs = Preferences.userNodeForPackage( this );

prefs.putInt( "windowX", windowX );

prefs.putInt( "window x", windowX );
prefs.putInt( "quiet please", windowX );
```

All that matters is that your application use the same key for reading and writing. A key can be any valid Java `String` with a length less than `Preferences.MAX_KEY_-LENGTH`, which is 80.

10.4.5 Allowable values

For the primitive Java types, any value is valid. String and byte array values must not be `null`.

In some key/value storage systems, storing a particular value is equivalent to removing that value entirely. This is not the case for the Preferences API. In keeping with the semantics of the `java.util.Hashtable` class, storing a value of `null` to a node will throw a `NullPointerException`.

Values must be smaller than `Preferences.MAX_VALUE_LENGTH` (8,192 bytes) in their stored representations. It can safely be assumed that the primitive types (`int`, `boolean`, `float`, `double`, and `long`) always have representations within this limit. String values can be compared against this limit using their `length()`

method, but remember that a Unicode character can require more than a single byte for its encoding.

Byte array values are encoded using the Base64 encoding as specified in RFC 2045, section 6.8, with one change. In practice, this means that the encoding of the byte array will be longer than the length of the byte array, as given by the expression `array.length`. According to the documentation, a byte array must be less than, or equal to, three-fourths of the value of `MAX_VALUE_LENGTH`. To determine whether a byte array is too long, use the following fragment:

```
if (array.length >= (MAX_VALUE_LENGTH*3/4)) {
  // ...
} else {
  // ...
}
```

Attempting to store a value that will not fit causes an `IllegalArgumentException` to be thrown.

10.4.6 Allowable node names

Node names must be smaller than `Preferences.MAX_NAME_LENGTH` (80 bytes) and cannot contain a "/" character. The node name of the root node is the empty string; no other node can have the empty string as a node name. These are the only restrictions on node names.

10.4.7 Default values

Each `get()` method takes a default value as the second argument. If the preferences database does not contain a value corresponding to the given key, or if the preferences database is not available, the default value is returned.

In the following example, we assume that the database does not contain a value corresponding to the key "`window x`":

```
Preferences prefs = Preferences.userNodeForPackage( this );
int windowX = prefs.getInt( "window x", 200 );
```

This code will result in the variable `windowX` containing the value `200`.

Default values are mandatory for *every* `get()` method. This is done to strongly encourage programmers not to assume that preferences will be available, and to ensure that their program works properly in such situations. The burden of creating reasonable defaults is placed on the programmer.

10.4.8 Removing values

Values can be removed entirely from the nodes that contain them by using the `remove()` method:

```
Preferences prefs = Preferences.userNodeForPackage( this );
prefs.putInt( "window x", 200 );
int windowX = prefs.getInt( "window x", 300 );
prefs.remove( "window x" );
int windowX = prefs.getInt( "window x", 300 );
```

Removing a value is not the same as putting an empty string or zero-valued integer there. Attempting to store a value of `null` to a preferences object will throw a `NullPointerException`.

The Preferences API also provides the `clear()` method. Calling this method is equivalent to calling `remove()` on all of the keys currently existing in the node:

```
Preferences prefs = Preferences.userNodeForPackage( this );
prefs.putInt( "window x", 200 );
int windowX = prefs.getInt( "window x", 300 );
prefs.clear();
int windowX = prefs.getInt( "window x", 300 );
```

Note that you do not need to specify the type of the value you are removing—the `remove()` method removes the value regardless of the type that was stored there.

10.4.9 Iterating through the values in a node

The method `keys()` returns an array containing the keys of all the key/value pairs in the node. This method can be used to iterate through the contents of a node:

```
Preferences uroot = Preferences.userRoot();
String keys[] = uroot.keys();
for (int i=0; i<keys.length; ++i) {
    System.out.println( keys[i]+" "+uroot.get( keys[i], "" ) );
}
```

Even though a node can contain both key/value pairs and child nodes, the `keys()` method only returns keys of key/value pairs. It does not return node names of child nodes.

10.4.10 Distinguishing between user and system nodes

As was mentioned in section 10.3.3, user and system nodes store information in the same way, and differ only in what they are used for and how they are accessible to different users. In fact, in most implementations, it's safe to assume that both kinds of nodes are implemented by the same Java class, which means you can't necessarily tell them apart by finding out what classes they are implemented by.

As a result, the Preferences API contains a method that allows you to distinguish between user and system nodes. This method is called `isUserNode()`, and it returns a boolean.

Here is an example of its use:

```
Preferences prefs = Preferences.userNodeForPackage( this );
if (prefs.isUserNode()) {
  // ...
} else {
  // ...
}
```

In this example, it is already obvious that the node is a user node, because of the call to `userNodeForPackage`. This will not always be the case, however—there will be some times when a node has been stored or passed to a method that does not know whether the node is a user node or a system node.

There aren't many situations where you would care what kind of node you had—and fewer still where, if you did care, you wouldn't know. You might care if you were writing utility code that dealt with preference data in some way, rather than just reading and writing preference data—such code might take a node as an argument to a method. If, for example, you wanted to impose a policy of only modifying values in user nodes, and treating system nodes as read-only, then you'd need to check the type before you wrote any data.

10.4.11 Node names and paths

Every node corresponds to an absolute path. This path can be accessed using the `absolutePath()` method. Each node also has a *node name* that corresponds to the last element in that node's absolute path, and that can be accessed via the `name()` method. (In the following code fragment, the output of each `System.out.println()` line is shown in bold italic following that line.)

```
Preferences root = Preferences.userRoot();
Preferences child1 = root.node( "child1" );
String child1Path = child1.absolutePath();
System.out.println( child1Path );
                // prints out:    /child1
String child1Name = child1.name();
System.out.println( child1Name );
                // prints out:    child1
```

The name and path values are meant to be reminiscent of a filesystem.

10.4.12 Getting parent and child nodes

A node that contains another node is called the *parent* of the contained node. The contained node is called the *child* of the containing node. For example, the relationship between parent and child nodes from figure 10.1 can be seen in table 10.4.

Table 10.4 Every node except the root node has a parent.

Parent	Child
/	/child0
/	/child1
/child1	/child1/grandchild0

The Preferences API provides the `parent()` and `childrenNames()` methods to allow you to get the parent or children of a given node.

Every node has exactly one parent, which is returned by the `parent()` method:

```
package a.b.c;
// ...
Preferences prefs = Preferences.userNodeForPackage( this );
Preferences parent = prefs.parent();
Preferences grandParent = parent.parent();
Preferences greatGrandParent = grandParent.parent();
Preferences greatGreatGrandParent = greatGrandParent.parent();
```

Actually, there's one exception: calling `parent()` on the root node returns `null`.

A node can have any number of children, and these children can be discovered via the `childrenNames()` method. A node that has no children will return a zero-length array (as opposed to `null`).

The following code fragment produces the array { `"child0", "child1"` }:

```
Preferences prefs = Preferences.userRoot();
String childrenNames[] = prefs.childrenNames();
```

In contrast, the following code fragment produces the zero-length array {}:

```
Preferences prefs = Preferences.userRoot().node( "child0" );
String childrenNames[] = prefs.childrenNames();
```

10.4.13 Determining the presence of nodes

You can determine whether a node is present within a preferences tree with the `nodeExists()` method. This method takes a string representing a path, which can be absolute or relative.

In the following code, calls to `nodeExists()` that return `true` are shown in boldface; those that return `false` are shown in italics.

```
Preferences uroot = Preferences.userRoot();
boolean b = uroot.nodeExists( "child1" );
b =         uroot.nodeExists( "/child1" );
Preferences child1 = uroot.node( "child1" );
Preferences grandchild0 = child1.node( "grandchild0" );
b = child1.nodeExists( "grandchild0" );
b = child1.nodeExists( "child1" );
b = child1.nodeExists( "/child1" );
b = grandchild0.nodeExists( "child1" );
b = grandchild0.nodeExists( "/child1" );
```

`nodeExists()` takes a single argument, which is an relative or absolute path. If the path is a relative path, the system attempts to locate the node relative to the given node. If the path is absolute, the system attempts to locate the node relative to the root of the given node; in this case, the only requirement for the node whose `nodeExists()` method is being called is that it be in the same preferences tree as the node in question.

10.4.14 Removing nodes

Child nodes can be removed from parent nodes by using the `removeNode()` method. The object that this method is called on is removed from the preferences tree, along with all of its descendants.

```
Preferences uroot = Preferences.userRoot();
Preferences child1 = uroot.node( "child1" );
child1.removeNode();
```

Note that after you've called the `removeNode()` method, you still have a reference to the object in question, which would allow you to continue to call methods on it, or any of its children. Calling any `Preferences` method on these objects, other than `name()`, `nodeExists()`, `flush()`, `isUserNode()`, or `absolutePath()` will result in an `IllegalStateException`. The code shown here in boldface is incorrect, and will throw an `IllegalStateException`:

```
Preferences uroot = Preferences.userRoot();
Preferences child1 = uroot.node( "child1" );
child1.removeNode();
// BAD!  Node has already been removed!
int windowX = child1.get( "window x", 200 );
// BAD!  Node has already been removed!
Preferences grandchild0 = child1.node( "grandchild0" );
```

It is impossible to remove the root node of a tree. Attempting to do so results in an `UnsupportedOperationException`.

10.4.15 Flushing

As mentioned in section 10.1.3, an implementation of the Preferences API is not required to write its data to the backing store immediately. Rather, it is allowed to simply make note of the write request, and then schedule a background process to do the writing at some later date.

In order to give the programmer some control over this mechanism, the Preferences API provides the `flush()` method:

```
int windowX = 250;
Preferences prefs = Preferences.userNodeForPackage( this );
prefs.putInt( "window x", windowX );
// ... changes are not permanent yet
prefs.flush();
// ... changes are now permanent
```

It's very important to understand that `flush()` only affects the backing store. The API itself, from the point of view of a Java program, will see any changes *immediately* after they are made. Flushing simply ensures that these changes have been safely saved to whatever medium is holding the permanent representation of the database. This medium is usually the local disk, but could also be a remote server or an intermittently available repository.

For the purposes of flushing, the addition and removal of nodes are considered regular changes—they do not become permanent until they are flushed.

Note that an implementation is not required to operate asynchronously. An implementation can choose to flush changes to the backing store after every single write, after only some writes, or at any other time. A program is not required to call `flush()` before program termination. This is taken care of automatically by the implementation.

10.4.16 Syncing

As mentioned in section 10.1.3, the Preferences API is intended to work in an environment where multiple agents are modifying the same backing store. These agents can be multiple threads within a Java Virtual Machine (JVM), or multiple JVMs, or programs written in different languages running on the same system.

Because of this, it is possible that changes made to the backing store by one thread or program will not be visible to another program. However, this other program can remedy the situation by calling the `sync()` method. This method ensures that the given node, and any of its descendants, are up to date with respect to the backing store. Any changes that have occurred to this node within the backing store will be reflected in this node after `sync()` returns.

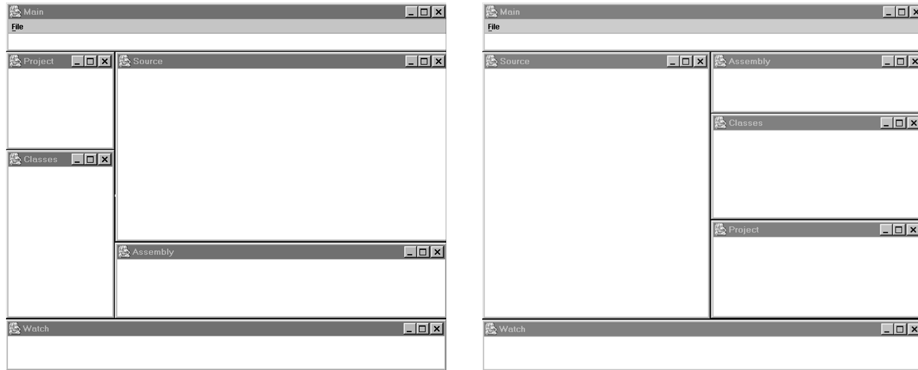

Figure 10.2 Two configurations of the interface to a hypothetical IDE application. Users generally need to customize the layout of such applications because there is so much information to see. The Preferences API is a perfect method for storing this customization data.

The following example demonstrates the interaction between two threads or programs reading from and writing to the same backing store:

```
Preferences prefs = Preferences.userNodeForPackage( this );
prefs.putInt( "window x", 300 );
int windowX = prefs.getInt( "window x", 250 ); // returns 300
// .. Long pause during which another thread or program
// sets this value to 500 in the backing store
windowX     = prefs.getInt( "window x", 250 ); // still returns 300
prefs.sync();
windowX     = prefs.getInt( "window x", 250 ); // returns 500
```

Note that `sync()` also ensures that any unflushed changes to this node within this JVM are flushed to the backing store. That is, `sync()` calls `flush()` before returning. It's not clear from the documentation which happens first—the `sync()` or the `flush()`.

10.4.17 Example: storing GUI configuration

A perfect use for the Preferences API is to store user customizations to the GUI of an application. In this example we'll look at the bare bones of the GUI for an integrated development environment (IDE). The interface of our hypothetical IDE has six windows.

IDEs generally have a very crowded interface. Because the interface is so crowded, users generally want to rearrange the windows to suit their needs (figure 10.2 shows two possible window arrangements for the IDE interface). These rearrangements should be saved, because it is very annoying for a user to have to set things up from scratch every time she runs the software.

Listing 10.1 contains a class called PersistentWindow, which knows how to store and retrieve its location and size information using the Preferences API. When a PersistentWindow is closed via its remove() method, or by clicking its close button, it saves its current state so that it can be restored when it is created again.

The PersistentWindows class creates all six windows and maintains a list of them. When the program is quit, the windows' remove() methods are called so that the state of the entire configuration is saved.

Listing 10.1 PersistentWindows.java

(See \Chapter10\org\jdk14tut\app\PersistentWindows.java)

```
package org.jdk14tut.app;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.util.prefs.*;
import javax.swing.*;

public class PersistentWindows
{
  private Set windows = new HashSet();
  private Preferences prefs =
    Preferences.userNodeForPackage( getClass() );
  static private final String NAMES[] = {
    "Main", "Source", "Assembly", "Classes", "Watch", "Project" };

  // Default positions of the windows
  static final private Rectangle DEFAULTS[] = {
    new Rectangle( 39, 2, 728, 81 ),
    new Rectangle( 218, 83, 548, 312 ),
    new Rectangle( 218, 396, 547, 125 ),
    new Rectangle( 39, 244, 179, 277 ),
    new Rectangle( 38, 522, 726, 85 ),
    new Rectangle( 39, 83, 179, 161),
  };

  public PersistentWindows() {
    setupGUI();
  }

  private void setupGUI() {                    ❶ Set up the windows
    PersistentWindow pw = null;

    pw = addWindow( 0 );
    pw.getContentPane().setLayout( new BorderLayout() );
    pw.getContentPane().add( new JTextArea(), BorderLayout.CENTER );
    JMenuBar mb = new JMenuBar();
    pw.setJMenuBar( mb );
    JMenu fileMenu = new JMenu( "File" );
```

```
  fileMenu.setMnemonic( KeyEvent.VK_F );
  mb.add( fileMenu );
  JMenuItem exitMI = new JMenuItem( "Exit", KeyEvent.VK_X );
  exitMI.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent ae ) {
        removeWindows();
        System.exit( 0 );
      }
    } );
  fileMenu.add( exitMI );
  pw.setVisible( true );

  pw = addWindow( 1 );
  pw.getContentPane().setLayout( new BorderLayout() );
  pw.getContentPane().add( new JTextArea(), BorderLayout.CENTER );
  pw.setVisible( true );

  pw = addWindow( 2 );
  pw.getContentPane().setLayout( new BorderLayout() );
  pw.getContentPane().add( new JTextArea(), BorderLayout.CENTER );
  pw.setVisible( true );

  pw = addWindow( 3 );
  pw.getContentPane().setLayout( new BorderLayout() );
  pw.getContentPane().add( new JTextArea(), BorderLayout.CENTER );
  pw.setVisible( true );

  pw = addWindow( 4 );
  pw.getContentPane().setLayout( new BorderLayout() );
  pw.getContentPane().add( new JTextArea(), BorderLayout.CENTER );
  pw.setVisible( true );

  pw = addWindow( 5 );
  pw.getContentPane().setLayout( new BorderLayout() );
  pw.getContentPane().add( new JTextArea(), BorderLayout.CENTER );
  pw.setVisible( true );
}

// Return the window with the given name
private PersistentWindow getWindow( String name ) {
  for (Iterator it=windows.iterator(); it.hasNext();) {
    PersistentWindow pw = (PersistentWindow)it.next();
    if (pw.name().equals( name ))
      return pw;
  }
  return null;
}

private PersistentWindow addWindow( int windowNum ) {
  PersistentWindow pw =
    new PersistentWindow( NAMES[windowNum], DEFAULTS[windowNum] );
```

2 Add a window, getting its location from Preferences

```
    windows.add( pw );

    return pw;
  }
```

3 Remove a window

```
  private void removeWindow( PersistentWindow pw ) {
    windows.remove( pw );

    // If there are no more windows left, quit
    if (windows.size()==0) {
      System.exit( 0 );
    }
  }
```

4 Remove all windows

```
  private void removeWindows() {
    Object ws[] = windows.toArray();
    for (int i=0; i<ws.length; ++i) {
      ((PersistentWindow)ws[i]).remove();
    }
  }

  /**
   * Inner class: PersistentWindow is a JFrame
   * whose position is managed by PersistentWindows
   * class.
   */
```

5 PersistentWindow represents a single window

```
  class PersistentWindow extends JFrame {
    private String name;

    public PersistentWindow( String name, Rectangle defaults ) {
      super( name );
      this.name = name;
      setLocation( defaults );
      addListeners();
    }

    public String name() { return name; }
```

6 Get location info from Preferences

```
    public void setLocation( Rectangle defaults ) {
      int x = prefs.getInt( name+"_x", defaults.x );
      int y = prefs.getInt( name+"_y", defaults.y );
      int width = prefs.getInt( name+"_width", defaults.width );
      int height = prefs.getInt( name+"_height", defaults.height );
      setLocation( x, y );
      setSize( width, height );
    }
```

Save location info to Preferences

```
    private void saveLocation() {
      int x = getLocation().x;
      int y = getLocation().y;
      int width = getSize().width;
      int height = getSize().height;
      prefs.putInt( name+"_x", x );
      prefs.putInt( name+"_y", y );
```

```
        prefs.putInt( name+"_width", width );
        prefs.putInt( name+"_height", height );
      }

      private void remove() {
        saveLocation();
        setVisible( false );

        // Remove this window from the parent
        // object's list
        removeWindow( PersistentWindow.this );
      }

      private void addListeners() {
        addWindowListener( new WindowListener() {
          public void windowActivated( WindowEvent we ) {
          }
          public void windowClosed( WindowEvent we ) {
          }
          public void windowClosing( WindowEvent we ) {
            // Remove window if the close-button is pressed
            remove();
          }
          public void windowDeactivated( WindowEvent we ) {
          }
          public void windowDeiconified( WindowEvent we ) {
          }
          public void windowIconified( WindowEvent we ) {
          }
          public void windowOpened( WindowEvent we ) {
          }
        } );
      }
    }

    static public void main( String args[] ) {
      new PersistentWindows();
    }
  }
```

7 Remove the window

1. Each window is initially created using `addWindow()`, which creates a `PersistentWindow`. Then each window is decorated with GUI components.

2. `addWindow()` creates a `PersistentWindow` and adds it to the list of windows. This list is maintained so that the program knows when all the windows have been closed, at which point the program can exit.

   The windows are indexed by an integer ranging from 0 to 5; this value serves as an index into the names array, which stores the name of each window, as well as into the table of default locations.

3. `removeWindow()` maintains the windows list. When there are no more windows, the program can quit.
4. `removeWindows()` calls the `remove()` method of each window. It must call `remove()` to ensure that the window saves its location information to the Preferences API.
5. `PersistentWindow` is a subclass of `JFrame` so that it can easily be used any place a `JFrame` is used. This makes it easy to use `PersistentWindows` throughout your application. Instead of having to explicitly remember the location of every window in your GUI, you can just remember to use a `PersistentWindow` for each window.
6. `setLocation()` attempts to load location and size information from the Preferences API. In the event that the preferences database is not available, it gets this info from the `defaults` object.
7. `remove()` calls `saveLocation()` before closing the window. After it closes the window, it calls the `removeWindow()` method of the parent object to make sure it gets removed from the list of windows.

This program illustrates a powerful technique: if each component can be responsible for its own customization information, it becomes easier to add such persistence to an application on an incremental basis. Programmers often avoid saving such customization data because it requires advance planning of a kind that generally isn't given a high priority. The Preferences API makes it very easy to make customizations persistent because it is globally available to all modules in an application, and does not require the assistance of the application's main module.

10.5 Change listeners

As was mentioned in section 10.1.3, multiple threads or programs can modify the same backing store, either through the Preferences API inside another JVM, or through another program written in another language, using a completely unrelated interface.

Because multiple agents can write to the same data store, it is desirable that a program be able to receive notice when data values change, and when values or nodes appear and disappear. To this end, the Preferences API provides *change listeners*. These allow your program to register its interest in receiving notice that certain changes have been made to the preferences database.

Note that agents outside the current JVM do not necessarily provide notice of changes they make. Because of the difficulty of implementing such a feature in a platform-independent way, an implementation of the Preferences API is only required to provide updates to changes that originate within the same API as the lis-

tener. However, implementations are encouraged to provide updates from other sources if possible.

A node can have any number of listeners attached to it.

10.5.1 Preference change listeners

A `PreferenceChangeListener` is an object that wants to receive notification when preference values are added, removed, or changed. Listeners are registered on a per-node basis, which means that a registration only ensures that the listener will receive notice of changes to that particular node.

The `PreferenceChangeListener` interface has a single method, `preferenceChange()`, which receives an object of type `PreferenceChangeEvent`:

```
Preferences prefs = Preferences.userNodeForPackage( this );

prefs.addPreferenceChangeListener( new PreferenceChangeListener() {
  public void preferenceChange( PreferenceChangeEvent pce ) {
    System.out.println( pce.getKey()+" <-- "+pce.getNewValue() );
  }
} );
prefs.putInt( "integer", 10 );
prefs.putInt( "integer", 20 );
prefs.remove( "integer" );
```

The preceding code in boldface produces the following output:

```
integer <-- 10
integer <-- 20
integer <-- null
```

Note the use of the `getNewValue()` method of the `PreferenceChangeEvent`. This method returns the new value of the preference node in question.

10.5.2 Node change listeners

A `NodeChangeListener` is like a `PreferenceChangeListener`, except that it is interested in changes to the node structure, rather than changes to the contents of nodes. Each time a child is added or removed from a node that has a listener, that listener is informed of the event via a `NodeChangeEvent` object.

```
Preferences prefs = Preferences.userNodeForPackage( this );
prefs.addNodeChangeListener( new NodeChangeListener() {
  public void childAdded( NodeChangeEvent nce ) {
    System.out.println( "Node added:\n\tparent="+nce.getParent()
                        +"\n\tchild="+
                        nce.getChild() );
  }
  public void childRemoved( NodeChangeEvent nce ) {
    System.out.println( "Node removed:\n\tparent="+nce.getParent()
```

```
                        +"\n\tchild="+
                        nce.getChild() );
  }
} );
Preferences abc = prefs.node( "a/b/c" );
Preferences a = prefs.node( "a" );
a.removeNode();
```

The preceding code in boldface produces the following output:

```
Node added:
     parent=User Preference Node: /<unnamed>
     child=User Preference Node: /<unnamed>/a
Node removed:
     parent=User Preference Node: /<unnamed>
     child=User Preference Node: /<unnamed>/a
```

Note that we don't get "node added" messages for nodes `b` and `c`—we haven't installed listeners on the parents of these nodes, so we don't know when they are added.

`NodeChangeEvent` provides `getParent()` and `getChild()` methods. These return the parent and child nodes of the operation that has occurred.

10.5.3 Example: listening for a GUI change request

Using listeners to respond to updates in the preference values is an excellent way to communicate customizations to different parts of a program. Normally, when a user customizes some aspect of an application, this change needs to be reflected immediately, and it needs to be reflected in all parts of the program that are affected by the change.

Taking care do this right can be tricky; doing it in full often inspires a listener-like system, so that it can be easy for any part of a program to *listen* for customizations. But why create your own listener-like system, when the Preferences API provides one? In this section, we'll consider a simple example of customization, and we'll use Preferences listeners to implement it.

The interface for our example program is simple. There are two windows. In the main window, there are two buttons that allow you to set the orientation of the secondary window. The two orientations are shown in figure 10.3.

When one of the buttons is clicked, it causes the secondary window to change its position and shape. However, it doesn't do this directly—it does this by setting a preference value. Meanwhile, the program has also registered a listener that listens for changes to this value. When the value change triggers the listener, the listener moves the window.

The advantage of this system is that the code that changes the preference value doesn't have to know which other objects or modules care about the value. Any module can watch for a value change by registering a listener. Additionally, changes made to the preference values from any other process trigger the same results as a change made from within the same process, allowing external tools to control your application.

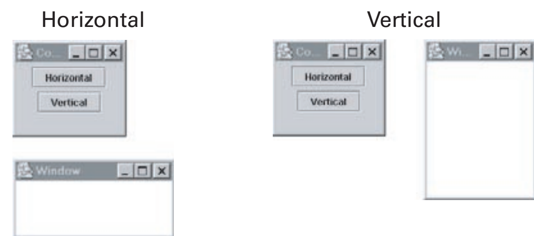

Figure 10.3 The secondary window changes position in response to clicking the buttons in the main window, and it uses listeners to do so.

The code for this example is shown in listing 10.2.

Listing 10.2 ListenerExample.java

(See \Chapter10\org\jdk14tut\app\ListenerExample.java)

```
package org.jdk14tut.app;

import java.awt.*;
import java.awt.event.*;
import java.util.prefs.*;
import javax.swing.*;

public class ListenerExample
{
  private JFrame window;
  private Preferences prefs =
    Preferences.userNodeForPackage( getClass() );
  static private final Rectangle horizontalOrientation =
    new Rectangle( 40, 220, 200, 100 );
  static private final Rectangle verticalOrientation =
    new Rectangle( 220, 40, 80, 200 );

  public ListenerExample() {
    addPrefsListener();
    setupGUI();
    setWindow();
  }

  private void addPrefsListener() {
    prefs.addPreferenceChangeListener(
      new PreferenceChangeListener() {
        public void preferenceChange( PreferenceChangeEvent pce ) {
          System.out.println( "Change: ("+pce.getNode()+") key="+
                              pce.getKey()+" value="+
                              pce.getNewValue() );
          setWindow();
```

● **Listen for a change to the value**

```
      }
    } );
  }

  private void setupGUI() {
    JFrame controlFrame = new JFrame( "Control" );
    JButton horizontal = new JButton( "Horizontal" );
    JButton vertical = new JButton( "Vertical" );
    Container cp = controlFrame.getContentPane();
    cp.setLayout( new FlowLayout( FlowLayout.CENTER ) );
    cp.add( horizontal, BorderLayout.NORTH );
    cp.add( vertical, BorderLayout.SOUTH );

    controlFrame.setLocation( 40, 40 );
    controlFrame.setSize( 120, 120 );

    window = new JFrame( "Window" );
    cp = window.getContentPane();
    cp.setLayout( new BorderLayout() );
    cp.add( new JTextArea(), BorderLayout.CENTER );

    horizontal.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent ae ) {
        setHorizontal();
      }
    } );

    vertical.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent ae ) {
        setVertical();
      }
    } );

    controlFrame.setVisible( true );
    window.setVisible( true );
  }

  private void setHorizontal() {
    prefs.putBoolean( "horizontal", true );
  }

  private void setVertical() {
    prefs.putBoolean( "horizontal", false );
  }

  private void setWindow() {
    boolean horizontal = prefs.getBoolean( "horizontal", true );
    Rectangle rect = null;
    if (horizontal) {
      rect = horizontalOrientation;
    } else {
      rect = verticalOrientation;
    }
```

Button clicks trigger calls to change Preferences value...

...here

Listener calls this method to move the window

```
    window.setVisible( false );
    window.setLocation( rect.getLocation() );
    window.setSize( rect.getSize() );
    window.doLayout();
    window.setVisible( true );
  }

  static public void main( String args[] ) {
    new ListenerExample();
  }
}
```

10.5.4 Example: changing server ports on the fly

Simple servers and daemons need to be restarted when their configuration changes; more sophisticated ones can change their configuration on the fly. This latter approach is a better solution for critical applications because it increases flexibility and reduces downtime—the less often you have to quit and restart, the better.

The Preferences API provides an excellent way for a server to respond to configuration changes. Not only does it allow configuration values to be changed at any time, it can potentially allow server administrators to use powerful tools to do so: since the preferences values are stored in a system-wide repository, the administrators can use available platform-specific tools to edit these values.

The example in this section uses a simple server called `Server` that listens on a port specified in preferences; additionally, changes to the preferences value cause the server to automatically switch to another port. This all happens within the running server—it does not need to be shut down and restarted.

As we've seen, this change of preference values can be initiated, on some platforms, by another process. However, since this feature is not supported on all platforms, Server uses a simple command-line shell, contained in the `CommandLine` and `ServerCommandLine` classes. There is only one command supported by this shell, which can be entered after you've started the program. The following fragment shows how the user uses the command-line interface—user-entered commands are in boldface:

```
java org.jdk14tut.app.Server
Listening on sun.nio.ch.ServerSocketChannelImpl[/0.0.0.0:5555]
port 5556
Change: (User Preference Node: /org/jdk14tut/app) key=port value=5556
Reopening....
Listening on sun.nio.ch.ServerSocketChannelImpl[/0.0.0.0:5556]
```

The second command, `port 5556`, tells the running server to switch from the current port (5555) to port 5556, which it does.

As in the example in section 10.5.3, this program does not respond *directly* to the user's command. Instead, the user's command sets a preference value, which triggers a listener. This means that the change will happen regardless of whether the preference value is changed from within this process, or, on platforms that support it, from another process.

Listing 10.3 shows the code for Server.java; listings 10.4 and 10.5 show `CommandLine` and `ServerCommandLine`, respectively.

Listing 10.3 Server.java

(See \Chapter10\org\jdk14tut\app\Server.java)

```
package org.jdk14tut.app;

import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.prefs.*;
import java.util.regex.*;

public class Server implements Runnable
{
  static private final int defaultPort = 5555;
  private int port;
  private Preferences prefs =
    Preferences.userNodeForPackage( getClass() );
  private ServerSocketChannel ssc;

  public Server() {
    port = getPort();
    addPrefsListener();
    new Thread( this ).start();
  }

  public Preferences prefs() {
    return prefs;
  }

  private int getPort() {
    int p = prefs.getInt( "port", defaultPort );
    return p;
  }

  private void addPrefsListener() {
    prefs.addPreferenceChangeListener(
      new PreferenceChangeListener() {
        public void preferenceChange( PreferenceChangeEvent pce ) {
          System.out.println( "Change: ("+pce.getNode()+") key="+
                              pce.getKey()+" value="+
                              pce.getNewValue() );
```

1 Get port number stored in Preferences

2 Listen for changes to prefs node

```
        if (pce.getKey().equals( "port" )) {
          try {
            updatePort();
          } catch( IOException ie ) {
            ie.printStackTrace();
          }
        }
      }
    } );
}

private void updatePort() throws IOException {
  int p = getPort();
  if (p != port) {
    changePort( p );
  }
}

private void changePort( int port ) throws IOException {
  this.port = port;
  ssc.close();
}

public void run() {
  try {
    while (true) {
      // Listen on port <port>, all addresses
      ssc = ServerSocketChannel.open();
      ssc.configureBlocking( true );
      byte anyIP[] = { 0, 0, 0, 0 };
      InetAddress localhost = InetAddress.getByAddress( anyIP );
      InetSocketAddress isa =
        new InetSocketAddress( localhost, port );
      ssc.socket().bind( isa );

      // Accept connections
      while (true) {
        try {
          System.out.println( "Listening on "+ssc );
          SocketChannel sc = ssc.accept();
          dealWithConnection( sc );

        } catch( AsynchronousCloseException ace ) {
          System.out.println( "Reopening...." );
          break;
        }
      }
    }
  } catch( IOException ie ) {
    ie.printStackTrace();
  }
}
```

3 **Respond to port change**

4 **Close ServerSocket Channel when port is changed**

5 **Listen for incoming socket connections**

6 **Process incoming connection**

7 **ServerSocketChannel is closed when the port is changed**

```
  protected void dealWithConnection( SocketChannel sc )
      throws IOException {
    System.out.println( "Got connection "+sc );
    sc.close();
  }

  static public void main( String args[] ) throws IOException {
    Server server = new Server();

    new ServerCommandLine( System.in, System.out );
  }
}
```

1. The initial port number is read from the Preferences API. Of course, when reading from Preferences, we need a default, which is what the `defaultPort` value (at the beginning of the code) is for.
2. We add a listener that listens to changes in preferences for this class. If a value change occurs, and it is a change in the value of the key `port`, then we call `updatePort()`, which will reconfigure the server.
3. `updatePort()` finds out what the new port value is, and then calls `changePort()` to reconfigure the server to the new value.
4. `changePort()` triggers a reconfiguration of the server by closing the `ServerSocketChannel` prematurely.
5. The `run()` method contains an infinite loop that does the traditional while-true-accept server inner loop, but with a twist: if the `ServerSocketChannel` is closed prematurely, this is taken as a signal to reopen on another port.
6. When a connection comes in, pass it to `dealWithConnection()`, which, of course, deals with it.
7. If the `ServerSocketChannel` is closed prematurely, this is taken as a signal to reopen on another port. This is done by simply exiting the inner `while()` loop, which sends us back to the top of the outer `while()` loop, where we prepare a new `ServerSocketChannel` on the new port.

The server changes ports in response to a command; the other two classes, `CommandLine` (see listing 10.4) and `ServerCommandLine` (see listing 10.5), implement this command-line interface. `CommandLine` is a generic command-line processing class, and `ServerCommandLine` is a subclass of `CommandLine` that knows how to set the server's port number.

Listing 10.4 CommandLine.java

(See \Chapter10\org\jdk14tut\app\CommandLine.java)

```
package org.jdk14tut.app;

import java.io.*;
import java.util.regex.*;

public abstract class CommandLine implements Runnable
{
  protected BufferedReader in;
  protected PrintWriter out;

  public CommandLine( InputStream in, OutputStream out ) {
    this( new InputStreamReader( in ),
          new OutputStreamWriter( out ) );
  }

  public CommandLine( Reader reader, Writer writer ) {
    this( new BufferedReader( reader ),
          new PrintWriter( writer ) );
  }

  public CommandLine( BufferedReader in, PrintWriter out ) {
    this.in = in;
    this.out = out;
    new Thread( this ).start();
  }

  public void run() {
    try {
      Pattern pattern = Pattern.compile( "\\s+" );
      while (true) {
        // Read each line and do simple parsing:
        // split the line on whitespace
        String line = in.readLine();
        if (line==null) {
          break;
        }

        String command[] = pattern.split( line );

        // Process each command
        boolean ok = processCommand( command );
        if (!ok) {
          System.out.println( "Unknown command: "+line );
        }
      }
    } catch( IOException ie ) {
      ie.printStackTrace();
    }
  }

  // Override this to implement commands
  abstract public boolean processCommand( String command[] );
```

CommandLine processes commands

Start a background thread to read

Parse each input line into white-space-separated strings

Pass the parsed command to processCommand()

Listing 10.5 ServerCommandLine.java

(See \Chapter10\org\jdk14tut\app\ServerCommandLine.java)

```
package org.jdk14tut.app;

import java.io.*;
import java.util.prefs.*;

public class ServerCommandLine extends CommandLine
{
  private Preferences prefs =
    Preferences.userNodeForPackage( getClass() );

  public ServerCommandLine( InputStream in, OutputStream out ) {
    super( in, out );
  }

  public boolean processCommand( String command[] ) {
    if (command[0].equalsIgnoreCase( "port" )) {
      int port = Integer.parseInt( command[1] );
      prefs.putInt( "port", port );
      System.out.println( "Set port to "+port );
      return true;
    } else {
      return false;
    }
  }

  static public void main( String args[] ) throws IOException {
   new ServerCommandLine( System.in, System.out );
  }
}
```

Use the same Preferences node that the server uses

Implement one command, port, which sets the port to the supplied value

This class can be used standalone (1)

(1) `Server.main()` starts a `ServerCommandLine`, which means that you can type `port` commands at the server's console. A `port` command looks like this:

```
port 5556
```

`ServerCommandLine.main()` also creates a `ServerCommandLine`, which means `ServerCommandLine` can be used as a standalone port-setting tool, setting the port from a separate JVM.

As was mentioned previously, changes made to a preferences value from one JVM might not trigger the `PreferenceChangeListeners` in another JVM; this depends on the implementation. Thus, the standalone `ServerCommandLine` tool may or may not trigger a running server to change ports—you'll have to try it to find out. If the standalone version doesn't cause the server to change ports, it will still store the new port value to the Preferences database.

It is important to note that the approach used here assumes that you will only want to run one copy of the server on your system. If you want to run multiple copies, then it makes sense to create Preferences nodes for each instance. The Preferences database then becomes a kind of instance storehouse, storing configuration values for instances while they run, and also when they are not running.

10.6 Stored defaults

The *stored defaults* feature of the Preferences API provides for a mechanism to supply values for keys that haven't been explicitly entered into the preferences database through normal channels. This allows defaults to be specified on a system-wide basis. These values will be present inside the preferences database, and will be returned by calls to the `get()` methods.

Note that the API does not provide methods for setting stored defaults; it is assumed that the backing implementation has its own method for setting these values. For example, the Preferences API might be implemented on top of some enterprise-wide directory service. In this case, the directory service itself must provide a way to set these values and expose them to the Java API as stored defaults.

When a program writes a value that was previously covered by an existing stored default, this default value is *overridden* by the new value. However, the stored default itself is not changed—it is merely shadowed by the value that was explicitly added.

When a key/value pair is removed for a key that has a stored default, this default value is *uncovered* by this removal. This means that rather than disappearing, the removed key takes on the value of the stored default. The behavior is the same for the `clear()` method.

10.7 Importing and exporting

Since the Preferences API is intended to work with a variety of platform-dependent backing stores, these backing stores will, in general, be incompatible with each other. In order to support the transfer of Preferences data from one system to another, the API provides `exportNode()`, `exportSubtree()`, and `importPreferences()` methods. These methods make use of an XML Document Type Definition (DTD) for the Preferences API, located at http://java.sun.com/dtd/preferences.dtd.

The `exportNode()` method exports the key/value pairs contained in a single node, but does not export any information about children of that node. `exportSubtree()` exports both the key/value pairs as well as information about child nodes. These child nodes, as well, are exported. The process is recursive, so that all descendants and their key/value pairs are exported at once.

The `importPreferences()` method imports the contents of a properly formatted XML file into the node it is called on. This file can contain any combination of key/value pairs and/or child nodes.

To give you an idea of what the preferences format looks like, here is code that dumps the contents of the tree shown in figure 10.1:

```
Preferences uroot = Preferences.userRoot();
Preferences child0 = uroot.node( "child0" );
Preferences child1 = uroot.node( "child1" );
Preferences grandchild0 = child1.node( "grandchild0" );
uroot.putInt( "integer", 10 );
child1.put( "name", "Greg" );
uroot.exportNode( System.out );
uroot.exportSubtree( System.out );
```

`exportNode()` results in the output shown in listing 10.6; `exportSubtree()` results in the output shown in listing 10.7.

Listing 10.6 Output of exportNode()

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE preferences SYSTEM 'http://java.sun.com/dtd/preferences.dtd'>

<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map>
      <entry key="integer" value="10" />     ● An integer is stored as a string
    </map>
  </root>
</preferences>
```

Listing 10.7 Output of exportSubtree()

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE preferences SYSTEM 'http://java.sun.com/dtd/preferences.dtd'>

<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map>
      <entry key="integer" value="10" />     ● An integer is stored as a string
    </map>
    <node name="child0">
```

```
      <map />
    </node>
    <node name="child1">
      <map>
        <entry key="bytes" value="AgMEBQ==" />
        <entry key="name" value="Greg" />
      </map>
      <node name="grandchild0">
        <map />
      </node>
    </node>
  </root>
</preferences>
```

A byte array is stored as a string in Base64 format

A string is stored as a string

XML is an excellent format for importing and exporting, since it is subject to rigorous standardization procedures and is ubiquitously supported.

10.8 Summary

The Preferences API is not just another way to sort small-scale data. It is a simple and flexible library for storing preference data that integrates directly with any system-wide preference system that may be available in the underlying operating system. By allowing for easy use of multiple data types and the possibility of stored defaults, the Preferences API is ideal for the kind of non-crucial data that applications need to store to enhance the user experience.