
# Getting Started Guide

Version 1.2 30 September 2021

Ciaran McHale

www.config4star.org

# Availability and Copyright

## Availability

The Config4* software and its documentation (including this manual) are available from www.config4star.org. The manuals are available in several formats:

- HTML, for online browsing.
- PDF (with hyper links) formatted for A5 paper, for on-screen reading.
- PDF (without hyper links) formatted 2-up for A4 paper, for printing.

## Copyright

AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# Contents

# Chapter 1

# Introduction

## 1.1 What is Config4*?

*Config4** (pronounced "config for star") is the generic name for a family of libraries that provides a powerful, easy-to-use parser for configuration files. Initially, this family has just two members: *Config4Cpp* (for C++) and *Config4J* (for Java). I hope that, over time, the family will grow to support other languages, including C and C#.

Many popular scripting languages—such as Lua, Perl, Python, Ruby and Tcl—are implemented in C. Thus, *Config4C* (for C) will be useful for C programmers, while also paving the way for scripting languages to parse Config4* configuration files. Likewise, one language supported on Microsoft's .NET platform can use functionality implemented in another .NET-supported language. Because of this, a port of Config4* to, say, C# would enable all .NET-based applications to use Config4*.

Hence the name of this overall project is *Config4**. The "*" in the name denotes a wildcard that can expand to include many different languages.

## 1.2 Why Might You Want to Use Config4*?

There are already many configuration technologies in widespread use. For example, Microsoft Windows provides a centralized registry; Java provides properties files; and it is common in scripting languages to store configuration information in the syntax of the scripting language. When

people need a platform- and programming language-agnostic configuration format, increasingly they choose XML. This raises the question: what has Config4* got to offer that isn't already provided by an existing configuration mechanism? The answer is: a lot.

### 1.2.1 Benefits for Users

Config4* offers several benefits for end users.

First, the syntax used in Config4* configuration files is much more user-friendly than, say, Java properties files or XML files.

Second, Config4* makes it easy for users to find and correct mistakes in configuration files. For example, Config4* provides a schema validator that can produce easy-to-understand error messages if configuration variables have misspelt names or bad values.

Third, users can choose the granularity they want for grouping configuration information.

- A user can have one configuration file for an application, split an application's configuration over several files, or have one file that contains configuration information for multiple (possibly related) applications.

- Config4* can *adapt* configuration information based on the environment in which an application is running. This means an administrator does not have to maintain multiple copies of an application's configuration file, where each copy has mostly common information but also a few details specific to a particular user, hostname or operating system. Instead, it is possible to maintain a *single* configuration file that contains the common information plus the details specific to each user, computer or operating system.

Fourth, Config4* can parse configuration information from an arbitrary source. Most commonly, a user will store configuration information in a file, but Config4* can obtain configuration information from, say, a web server or a database. This provides the option of centralizing configuration information if you are deploying an application on several computers that do not have access to a shared file system.

### 1.2.2 Benefits for Developers

Config4* offers several benefits for developers.

First, the Config4* programming API is not easy to use; it is *trivial* to use.

Second, Config4* provides a utility that makes it possible to compile a configuration file into an application's executable. This is useful for embedded systems, or where you want to embed default configuration values into an application so the use of an external configuration file becomes optional.

Third, Config4* provides a viable alternative to XML as a data file format. The Config4* library is an order of magnitude smaller than popular XML parser libraries, it parses input files speedily and the resulting in-memory representation is more compact than a DOM tree. In addition, Config4* provides user-friendly error messages, easy-to-use schema validation of input, and a programmer-friendly API.

Finally, one of the goals of the Config4* project is to provide highly portable implementations for many popular programming languages. This is important because many projects start out by using just a single programming language on one operating system but, over time, end up using several languages and/or operating systems. For example, a client-server system might initially be implemented in C++. Then you decide to re-implement the client part in Java so it can provide a cross-platform graphical user interface (GUI). Then you use a scripting language to write some report-generating utilities for the system. In such an environment, being able to process one set of configuration files from multiple languages is a great help.

## 1.3 The Collection of Config4* Manuals

Config4* documentation is provided as a collection of manuals.

***Config4* Getting Started Guide.*** This is the manual you are currently reading. It provides an introduction to Config4* for both users and developers of Config4*-based applications.

***Config4* Practical Usage Guide.*** This manual provides useful advice and suggestions for developers on practical ways to use Config4* in a wide variety of projects.

***Config4* Maintenance Guide.*** This manual provides useful background information for people who want to maintain the source code of Config4* implementations or implement Config4* in another programming language.

***Config4* C++ API Guide.*** This manual provides a tutorial and reference for C++ programmers.

***Config4* Java API Guide.*** This manual provides a tutorial and reference for Java programmers.

## 1.4 Structure of this Manual

This manual is structured as follows.

Part I provides an overview of Config4*'s capabilities for both users and developers.

Part II discusses Config4*'s supporting infrastructure, including its security mechanism and command-line utilities.

Part III provides provides comprehensive details on the syntax used in Config4* files and also the syntax used in its schema language.

Part IV provides tips on how to use Config4* in your projects.

## 1.5 Obtaining and Installing Config4*

The main website of Config4* is www.config4star.org. Some other domains (www.config4cpp.org and www.config4j.org) redirect to the main website.

You can download Config4Cpp and Config4J from the Config4* website. The downloads are in the form of a ZIP file. The downloads contain C++ or Java source code plus manuals (in PDF and LaTeX form).

You should read the top-level `README.txt` file in a distribution for instructions on how to compile the Config4* source code.

# Part I

# Overview

# Chapter 2

# Overview of Config4* Syntax

This chapter provides an overview of the syntax used in Config4* configuration files. A complete definition of the syntax is provided in Chapter 8.

## 2.1 Comments, Variables and Scopes

Figure 2.1 provides a simple example of a Config4* configuration file.

Comments, like the one shown in line 1, start with `"#"` and continue until the end of the line. Most of the lines in a configuration file contain assignment statements. These are of the form *name*=*value*, where the *value* can be a string (line 2) or a list of strings (line 4). You can use the `"+"` operator to concatenate both strings (line 3) and lists (line 6). Strings are usually delimited between double quotes.

There are two ways to write a string. The first way (which is illustrated in Figure 2.1) is as a sequence of characters enclosed within double quotes. Within such a string, `"%"` acts as an escape character. For example, `%n` denotes a newline character, and `%"` denotes a double quote.

The second way to write a string is as a (possibly multi-line) sequence of characters enclosed between `<%` and `%>`. No escape sequences are recognised between `<%` and `%>`. The `<%...%>` notation is useful if you want to embed, say, a code segment in a configuration file. You can combine both forms of string by using the string concatenation (`"+"`) operator.

Figure 2.1: Example configuration file

```
 1  # this is a comment
 2  name = "Fred";
 3  greeting = "hello, " + name;
 4  some_names = ["Fred", "Mary", "John"];
 5  more_names = ["Sue", "Ann", "Kevin"];
 6  all_names = some_names + more_names;
 7  server.defaults {
 8      timeout = "2 minutes";
 9      log {
10          dir = "C:\foo\logs";
11          level = "0";
12      }
13  }
14  foo_srv {
15      @copyFrom "server.defaults";
16      log.level = "1";
17  }
18  bar_srv {
19      @copyFrom "server.defaults";
20      timeout = "30 seconds";
21  }
```

A configuration file can contain named scopes (lines 7, 9, 14, and 18 in Figure 2.1). Scopes can be nested (line 9) and re-opened. The scoping operator is `"."`. For example, the name `log.level` refers to a variable called `level` inside a scope called `log`. You do not have to explicitly open a scope to define a variable or a nested scope within it. For example, line 7 opens the `server.defaults` scope without opening the outer `server` scope. Likewise, line 16 defines `log.level` without explicitly opening the `log` scope.

## 2.2 Copying Default Values

All keywords (for example, `@include`, `@if` and `@copyFrom`) start with the `"@"` symbol: this ensures there can never be a clash between the name of a keyword and the name that you might wish to use for a configuration variable or scope.

The `@copyFrom` statement (lines 15 and 19 in Figure 2.1) copies the entire contents (variables and nested scopes) of the specified scope into

the current scope. This provides a simple, yet effective, reuse mechanism. For example, if several applications use similar configuration values then you can put common values into one scope and then use the `@copyFrom` statement to copy these into application-specific configuration scopes. It is *not* an error to assign a new value to an existing variable. This makes it possible to override default values obtained via a `@copyFrom` statement.

## 2.3 Including Other Files

An `@include` statement (not shown in Figure 2.1) includes the contents of another configuration file into the current one. For example:

```
@include "/tmp/foo.cfg";
```

You can use string concatenation to form the file name. For example:

```
@include fileToDir(configFile()) + "/subsystem1.cfg";
@include fileToDir(configFile()) + "/subsystem2.cfg";
@include fileToDir(configFile()) + "/subsystem3.cfg";
```

This example also uses `fileToDir(configFile())`, which is a combination of two built-in function calls that returns the name of the directory in which the configuration file being parsed resides. This technique of combining the `@include` command with these built-in functions enables you to split a (potentially) large amount of configuration information across several smaller files. Doing this can simplify the maintenance of configuration files.

Config4* has many built-in functions. You can find a complete list of them in Section 8.12 on page 91. In Section 2.2, I mentioned that all keywords are prefixed with `"@"` to prevent the possibility of a clash between a keyword and the name that you might wish to use for a configuration variable or scope. For the same reason, all functions are suffixed with `"("`. Thus, `fileToDir(` is the start of a function call, but `fileToDir` is the name of a variable or scope.

## 2.4 Including the Output of Commands

The `@include` command can include not just files, but also the output resulting from executing arbitrary shell commands. For example, the `curl` utility (http://curl.haxx.se) is a command-line tool that can output the contents of a specified URL, such as a web page or a file at an FTP

site. If you have `curl` installed on your computer, then a configuration file can have `@include` commands similar to those shown below.[1]

```
@include "exec#curl -sS http://localhost/someFile.cfg";
@include "exec#curl -sS ftp://localhost/someFile.cfg";
```

As these examples illustrate, if the argument to an `@include` statement starts with `"exec#"` then the argument is executed as a shell command and the standard output from that command is included.

The ability to execute arbitrary commands is very flexible, but it poses a security risk. For example, we need to guard against a malicious person adding something like the following to a configuration file on Windows.

```
@include "exec#del /F /S /Q C:\";
```

Such a command would delete everything on the C: drive of the computer (somewhat similar to `"exec#rm -rf /"` on UNIX). Chapter 5 discusses the mechanism that Config4* provides to guard against such security threats.

## 2.5 Accessing the Environment

You can access environmental information in a configuration file. For example, you can use `getenv("FOO_HOME")` to access an environment variable called `FOO_HOME`.

```
install_dir = getenv("FOO_HOME");
```

You can use the `replace()` function to perform a search-and-replace, as the example below demonstrates.

```
install_dir = replace(getenv("FOO_HOME"), "\", "/");
```

In the above example, the `replace()` function replaces all occurrences of `"\"` with `"/"` in the specified string. This is a useful tactic when you run an application on Windows that insists on dealing with UNIX-style file and directory names.

You can use the `exec("command")` function to execute an external command and capture its standard output. For example, on both UNIX and Windows, the `hostname` command prints the name of the computer. You can access this information as shown in the following example:

[1] By default, `curl` prints diagnostics to standard error. The `-s` option instructs `curl` to be silent, but unfortunately, this option means that `curl` does not print error messages either. You can use `-sS` to instruct `curl` to print error messages but no other diagnostics.

```
url = "http://" + exec("hostname") + ":8080/"
log_dir = "/net/" + exec("hostname") + "/logs";
```

## 2.6 Temporary Variables

Sometimes you may want several variables to have values that share a common prefix. Rather than explicitly (re)stating the common prefix several times, you might decide to assign it to a temporary variable, use that temporary variable to help you define the "real" variables, and then finally `@remove` the temporary variable. The example below illustrates this.

```
_install_dir = getenv("FOO_HOME");
bin_dir = _install_dir + "/bin";
etc_dir = _install_dir + "/etc";
log_dir = _install_dir + "/logs";
@remove _install_dir;
```

As the above example illustrates, a convention is that the name of a temporary variable starts with an underscore. The `@remove` statement does what its name suggests: it removes the specified configuration variable.

You may wonder what is the point of removing a variable: why not just leave `_install_dir` in existence? The answer is that by insisting a configuration file contain *only* required variables, an application can make use of a schema validator that can perform extensive error checking on the contents of a configuration file. I will discuss schema validation later (Section 3.10 on page 31).

## 2.7 The `@if`-then-`@else` Statement

By themselves, the `exec()` and `getenv()` functions (discussed earlier in this chapter) are of limited use. However, they become much more useful when combined with an `@if`-then-`@else` statement. You can see some examples of this is in Figure 2.2.

To reduce the chances of a mis-configured client application on a test machine accidentally communicating with a server application in production, some organizations use one set of server port numbers in testing, and a different set of server port numbers in production. Traditionally, this separation was accomplished by having one configuration file for test machines, and having another configuration file for production machines. However, the cascading `@if`-then-`@else` statement at lines 4–14

Figure 2.2: Configuration file with advanced features

```
 1  production_hosts = ["pizza", "pasta", "zucchini"];
 2  test_hosts       = ["foo", "bar", "widget", "acme"];
 3
 4  @if (exec("hostname") @in production_hosts) {
 5      server_x.port = "5000";
 6      server_y.port = "5001";
 7      server_z.port = "5002";
 8  } @elseIf (exec("hostname") @in test_hosts) {
 9      server_x.port = "6000";
10      server_y.port = "6001";
11      server_z.port = "6002";
12  } @else {
13      @error "This is not a production or test machine";
14  }
15  if (osType() == "windows") {
16      tmp_dir = replace(getenv("TMP"), "\", "/");
17  } @else {
18      tmp_dir = "/tmp";
19  }
```

in Figure 2.2 shows it is possible to have a single configuration file that adapts itself to its environment.

The `@error` statement (line 13) instructs Config4* to stop parsing and instead report an error. This provides a way for a configuration file to report that it is being used outside of its intended domain.

The `osType()` function (line 15) returns a string, such as `"windows"` or `"unix"`, that indicates the host operating system. If you want to check which variant of UNIX is being used then you can use `exec("uname")`.

## 2.8 Conditional `@include` and `@copyFrom`

By default, `@include` reports an error if the specified file does not exist. However, if you place `@ifExists` at the end of an `@include` statement, then `@include` does not complain about a non-existent file.

```
@include "/path/to/foo.cfg" @ifExists;
```

The conditional `@include` provides a way for an application's configuration file to set default values and then include an optional user-specific configuration file to override default values. For example, the configura-

tion file for a program called `foo` running on UNIX might be structured as shown below.[2]

```
# Set default configuration values
...
# Now optionally include user-specific overrides
@include getenv("HOME") + "/.foo.cfg" @ifExists;
```

You can use an `"@ifExists"` clause not just with an `@include` statement, but also with `@copyFrom`, as shown below.

```
override.pizza { ... }
override.pasta { ... }
foo_srv {
   # Set default values
   ...
   # Modify some values for particular hosts
   @copyFrom from "override." + exec("hostname") @ifExists;
}
```

## 2.9 Append Assignment

The *append assignment* statement, which uses the `"+="` operator, was introduced in version 1.2 of Config4*.

```
greeting = "Hello";
greeting += ", world";
```

The second line in the above example is equivalent to the line below.

```
greeting = greeting + ", world";
```

The append assignment statement is often used in conjunction with the `"@copyFrom"` command, as shown below.

```
app.defaults {
  options = ["default", "options"];
  ...
}
my_app {
  @copyFrom "app.defaults";
  options += ["extra options"];
}
```

[2] In UNIX, the `HOME` environment variable specifies the "home" directory for a user, which is where a user normally stores personal files. By convention, the name of a configuration file for an application stored in this directory starts with `"."`, and is followed by the name of the application.

## 2.10 Conditional Assignment

Config4* provides a way for an application to integrate command-line options with a configuration file. To illustrate this, consider an application that is started in the following manner.

```
myApp.exe -set username Fred -set password fgTR742 -cfg foo.cfg
```

The application could be written to perform the following steps during initialisation.

1. The application creates an (initially empty) configuration object.
2. The application examines its command-line options. Whenever it encounters an option of the form "`-set` *`name value`*", it inserts that *`name-value`* pair to the configuration object.
3. Finally, the application uses the configuration object to parse the file specified by the "`-cfg` *`file`*" command-line option.

The above algorithm ensures that the command-line options processed in step 2 become "preset" variables in the configuration object when the configuration file is parsed (step 3).

Within a configuration file, the `?=` operator performs *conditional assignment*; it assigns a value to a variable only if the variable does not already have a value.

```
username ?= "";
password ?= "";
```

In this way, a configuration file can provide default values for some variables, and those default values can be overridden via command-line options on the application.

## 2.11 Centralizable and Adaptive Configuration

Some basic capabilities of Config4*, for example, *name=value* pairs and scopes, can be found in other configuration technologies. However, many of its other capabilities are not so common.

- You can use `getenv()` to access a named environment variable, such as `HOME` or `USERNAME`. You can also use `osType()` to determine the operating system's type.

- You can use `exec()` to capture the output from executing an external command, such as `hostname` or (on UNIX) `uname`.

- You can pass the results of `getenv()`, `exec()` or `osType()` as arguments to `@include` or `@copyFrom` statements, or use them in conditions in `@if`-then-`@else` statements.

These capabilities mean that *one* Config4* file can contain configuration for *multiple* users, running an application on *multiple* computers and *multiple* operating systems. Or to put it another way: a configuration file can "adapt" itself to its environment. I call this ability *adaptive configuration*.

The ability of Config4* to parse not just a configuration file but also the output of external commands, such as `curl`, makes it possible for an organization to *centralize* the adaptive configuration files of Config4*-enabled applications. Such centralization can significantly reduce administration overheads, especially when a large organization deploys an application on hundreds, thousands, or even tens of thousands, of computers.

## 2.12 The `uid-` prefix

The discussion in this chapter so far has focussed on using Config4* to store configuration information, which, in essence, is simple data in the form of *name=value* pairs, optionally organised into scopes. In this section, I discuss an additional feature of Config4* that makes it possible to store more complex data in Config4* files, thus greatly expanding the potential range of uses of Config4*.

Let's assume you want to store some information about employees in a configuration file. You might try writing the following.

```
employee { name = "John Smith"; ... }
employee { name = "Jane Doe"; ... }
```

However, that will not work. This is because the second occurrence of the `employee` scope re-opens the existing scope, so the details of Jane Doe overwrite those of John Smith. You could work around this by using a unique number as a suffix on the name of each scope.

```
employee_1 { name = "John Smith"; ... }
employee_2 { name = "Jane Doe"; ... }
```

This will work, but you have to keep track of the numbers that have been used already to ensure you do not accidentally reuse one of those numbers in the name of a new scope. Config4* eliminates this burden by treating an *identifier* (that is, the name of a scope or variable) in a special way if it starts with `"uid-"`; *uid* is an abbreviation for *unique identifier*. Consider the following file.

```
uid-employee { name = "John Smith"; ... }
uid-employee { name = "Jane Doe"; ... }
```

Config4* keeps a counter that starts at zero and is incremented for each identifier starting with `"uid-"`. Config4* automatically renames these identifiers so that the counter (expressed as a nine-digit number) is embedded in them. For example, the first occurrence of `uid-employee` might be renamed as `uid-000000000-employee`, the next occurrence renamed as `uid-000000001-employee`, the next occurrence renamed as `uid-000000002-employee` and so on.[3]

You might be wondering why the unique number is always expressed as nine digits with leading zeros. The reason has to do with how Config4* is implemented. When Config4* parses a configuration file it stores all the *entries* (that is, variables and scopes) in hash tables. Hash tables provide a fast lookup mechanism but they do not preserve the order in which the entries were originally defined in the input file. However, the API of Config4* makes it easy for a program to get a sorted list of entries. Expressing uid numbers as nine digits with leading zeros guarantees that a sorted list of entries contains all the uid entries in the order in which they appeared in the input file. This makes it possible for a program to process uid entries in their original order, if desired.

As a slightly contrived example for the use of uid entries, consider a file that stores recipes, like that in Figure 2.3. Each recipe is stored in its own `uid-recipe` scope. I do not care about the order of recipes, but the `"uid-"` prefix frees me from the burden of having to think of a unique name for the scope of each recipe. Within a `uid-recipe` scope, the relative order of the `ingredients` and `name` entries is not important so they do not have a `"uid-"` prefix. However, each step in the recipe must be performed in strict sequence so they have a `"uid-"` prefix.

[3] The use of a nine-digit number means that Config4* can cope with up to $10^9$ uid entries. This number is what most English-speaking countries call a *billion*, but many other countries call a *thousand million* (and they use the term *billion* to mean $10^{12}$, that is, a *million million*): http://en.wikipedia.org/wiki/Long_and_short_scales. In the extremely unlikely event that you exceed the limitation of $10^9$ uid entries, the *Config4* Maintenance Guide* explains how you can make simple changes to the source code of Config4* to increase the limit.

Figure 2.3: File of recipes

```
uid-recipe {
    name = "Tea";
    ingredients = ["1 tea bag", "cold water", "milk"];
    uid-step = "Pour cold water into the kettle";
    uid-step = "Turn on the kettle";
    uid-step = "Wait for the kettle to boil";
    uid-step = "Pour boiled water into a cup";
    uid-step = "Add tea bag to cup & leave for 3 minutes";
    uid-step = "Remove tea bag";
    uid-step = "Add a splash of milk if you want";
}
uid-recipe {
    name = "Toast";
    ingredients = ["Two slices of bread", "butter"];
    uid-step = "Place bread in a toaster and turn on";
    uid-step = "Wait for toaster to pop out the bread";
    uid-step = "Remove bread from toaster and butter it";
}
```

Although most readers will not be interested in using Config4* to store recipes, the issues I described in that example often occur in real-world systems. A typical case is Ant (http://ant.apache.org), which is a popular build system for Java-based applications (in much the same way that `make` is a popular build system for C and C++ applications). Ant reads a build specification from an XML file. The build file contains, among other things, a collection of `target` elements that are analogous to a "recipe" for compiling or packaging a unit of software. Within each `target` element there is an ordered collection of tasks, which are analogous to the ordered "steps" within a recipe. A target may also have a list of targets upon which it depends; in Config4* this could be expressed as a non-uid variable, similar to `ingredients` in Figure 2.3.

## 2.13 Summary

Config4* has several features, such as *name=value* pairs, scopes and an `@include` statement, that are common to many other configuration technologies. However, Config4* provides additional capabilities that are more rare, and which are very useful.

- *Adaptable configuration.* A Config4* file can use `getenv()`, `exec()`

and `osType()` to query its environment, and the results of these queries can be used in `@if`-then-`@else`, `@include` and `@copyFrom` statements. This enables a configuration file to adapt to its environment. In addition, conditional assignment (the `?=` operator) enables a configuration file to take account of command-line arguments.

- *Centralised configuration.* Config4* can parse not just a configuration file, but also the output of executing a command. Combining this capability with `curl` makes it feasible to store a configuration file in a centralised location, such as a web server. Such centralization can significantly reduce administration overheads, especially when a large organization deploys an application on hundreds or thousands of computers.

- *Uid* entries. The `"uid-"` prefix makes it possible for Config4* to be used to store not just simple configuration files but also complex, structured data in which there may be multiple items of a similar nature or guaranteed ordering of items is important.

This chapter has presented an overview of the syntax used in a Config4* configuration file (you can find full details in Chapter 8). The next chapter provides an overview of the API provided by Config4* for C++ and Java programmers.

# Chapter 3

# Overview of the Config4* API

## 3.1 Introduction

The C++ and Java APIs of Config4* are very similar, so this chapter discusses both of them side by side. All the functionality of Config4Cpp is defined in the `config4cpp` namespace. The functionality of Config4J is defined in the `org.config4j` package. To illustrate the API of Config4*, consider a configuration file that contains the following entries.

```
foo_srv {
    timeout ?= "2 minutes";
    log {
        dir ?= "C:\foo\logs";
        level ?= "0";
    };
};
```

Figures 3.1 and 3.2 show examples of using Config4Cpp and Config4J to access information in the above configuration file. In much of this chapter I discuss the APIs used in these figures.

The correct behaviour Config4Cpp depends on the locale being set correctly. Because of this, it is advisable to call `setlocale()` *before* invoking any Config4Cpp APIs. If you do this, then Config4Cpp will be able to handle characters defined in your locale, such as European accented characters or Japanese ideographs. If you neglect to call `setlocale()`,

Figure 3.1: Example of Using Config4Cpp

```
#include <locale.h>
#include <config4cpp/Configuration.h>
using namespace config4cpp;
...
setlocale(LC_ALL, "");
...
const char *    logDir;
int             logLevel, timeout;
const char *    scope = "foo_srv";
Configuration * cfg = Configuration::create();
try {
  cfg->parse(getenv("FOO_CONFIG"));
  logDir   = cfg->lookupString(scope, "log.dir");
  logLevel = cfg->lookupInt(scope, "log.level");
  timeout  = cfg->lookupDurationSeconds(scope, "timeout");
} catch(const ConfigurationException & ex) {
  cout << ex.c_str() << endl;
}
cfg->destroy();
```

Figure 3.2: Example of Using Config4J

```
import org.config4j.*;
...
String        logDir;
int           logLevel, timeout;
String        scope = "foo_srv";
Configuration cfg = Configuration.create();
try {
  cfg.parse(cfg.getenv("FOO_CONFIG"));
  logDir   = cfg.lookupString(scope, "log.dir");
  logLevel = cfg.lookupInt(scope, "log.level");
  timeout  = cfg.lookupDurationSeconds(scope, "timeout");
} catch(ConfigurationException ex) {
  System.out.println(ex.getMessage());
}
```

then Config4Cpp is likely to correctly process *only* characters in the 7-bit US ASCII character set.

## 3.2 Parsing Configuration Files

You create a configuration object by invoking the static `create()` operation on the `Configuration` class. The newly created configuration object is empty initially. You can populate it by invoking the `parse()` operation, which takes a file name as a parameter. The C++ example (Figure 3.1) calls the `getenv()` function to obtain the file-name parameter from an environment variable. For part of Java's history, it was difficult to access environment variables in Java applications but Config4J provides a utility `getenv()` operation on the `Configuration` class to simplify such access.[1]

If `parse()` encounters any errors, then it throws an exception of type `ConfigurationException`. The C++ implementation of this class provides a `c_str()` operation you can use to access the exception's message. Java developers can access the exception's message in the usual Java way, that is, by calling `getMessage()`. In Java, `ConfigurationException` is a runtime exception.

## 3.3 Accessing Configuration Variables

Once a `Configuration` object has been created and populated, you can use operations such as `lookupString()` and `lookupList()` to retrieve the values of configuration variables. You can see examples of this in Figures 3.1 and 3.2.

Some additional operations with names of the form `lookup<Type>()` are provided that retrieve a string value and convert it to another datatype. For example, `lookupInt()` converts a string value to an integer and `lookupBoolean()` converts a string value to a boolean.

The `lookupDurationSeconds()` operation converts strings, for example, `"10 seconds"` or `"2.5 minutes"`, into an integer that denotes the duration in seconds (it converts `"infinite"` to the integer value -1). You can use such durations to configure timeout values in applications. There are also `lookupDurationMilliseconds()` and `lookupDurationMicroseconds()` operations in case you prefer to have the result expressed in milliseconds or microseconds rather than in seconds.

If a lookup operation fails—for example, `lookupInt()` might encounter an invalid integer—then it throws a `ConfigurationException`. The mes-

[1] Section 4.2 on page 37 explains why, before Java 1.5, it was difficult to access environment variables in Java, and how Config4J works around this difficulty.

sage contained in the exception explains what went wrong.

## 3.4 Scoped Names

Some Config4* operations take two parameters that, when combined, specify the fully-scoped name of a configuration variable. For example, in C++, you can access the value of `foo_srv.log.dir` with the following statement.

```
logDir = cfg->lookupString("foo_srv", "log.dir");
```

The example code in Figures 3.1 and 3.2 illustrates the intended purpose of this approach to identifying configuration variables. A variable, called `scope`, is initialized with the name of a configuration scope, and a configuration variable (such as `log.dir`) within that scope can be accessed by passing `scope` and the name of the variable as parameters to an accessor operation.

```
logDir = cfg->lookupString(scope, "log.dir");
```

Typically, the `scope` variable is obtained from a command-line argument. By rerunning an application with a different command-line argument, you can change the scope used to configure the application. For example, you might have one configuration scope for running an application *without* debugging diagnostics, and another scope that *enables* debugging diagnostics. Alternatively, you might have a separate scope for each user or for each instance of a replicated server application.

## 3.5 Presetting Configuration Variables

When Config4* is parsing a configuration file, it calls `insertString()` and `insertList()` to populate the `Configuration` object with name-value pairs. You can call those operations directly in your application code. One important reason for doing so is to populate a `Configuration` object with name-value pairs obtained from command-line arguments *before* parsing a configuration file. The Java code in Figure 3.3 illustrates how to do this.

This tactic provides a simple way to integrate command-line options with information in a configuration file. To understand why, consider the configuration file shown at the start of this chapter, which is repeated below for convenience.

Figure 3.3: Java example of presetting configuration variables

```
public void main(String[] args) {
  String        logDir;
  int           logLevel, timeout;
  String        scope = "foo_srv";
  Configuration cfg = Configuration.create();
  try {
    //--------
    // Pre-populate the configuration object from
    // "-set name value" command-line options
    //--------
    for (int i = 0; i < args.length; i++) {
      if (args[i].equals("-set") {
        if (i + 2 >= args.length) {
          usageError("Too few arguments after '-set'");
          System.exit(1);
        }
        cfg.insertString(scope, args[i+1], args[i+2]);
      } else {
        ... // processing for other command-line options
      }
    }

    //--------
    // Parse the config file and lookup config variables.
    //--------
    cfg.parse(cfg.getenv("FOO_CONFIG"));
    logDir   = cfg.lookupString(scope, "log.dir");
    logLevel = cfg.lookupInt(scope, "log.level");
    timeout  = cfg.lookupDurationSeconds(scope, "timeout");
  } catch(ConfigurationException ex) {
    System.out.println(ex.getMessage());
    System.exit(1);
  }
}
```

```
foo_srv {
    timeout ?= "2 minutes";
    log {
        dir ?= "C:\foo\logs";
        level ?= "0";
    };
```

```
};
```

The use of the conditional assignment operator (`"?="`) within the configuration file means that a variable will be assigned a value only if it does not already have a value. For example, running the code shown in Figure 3.3 with the command-line option `"-set log.level 2"` will change the log level from its default value of 0 to the value of 2.

## 3.6 Variations of `parse()`

Earlier in this chapter (in Section 3.2 on page 21) I said that you can call `parse()` to parse a configuration file. Actually, Config4* offers a lot of flexibility in parsing, as I now discuss.

### 3.6.1 Parsing Centralized Configuration

Let's assume that, as shown in Figures 3.1 and 3.2, an application uses the `FOO_CONFIG` environment variable to specify the location of its configuration file. If the application is being used only by you and on only one computer then you can store the application's configuration information in a file and set `FOO_CONFIG` to point to this.

```
FOO_CONFIG=/path/to/foo.cfg
```

A few months later you may want to use the application on *several* computers within the same office. You *could* copy the configuration file onto each of these computers but then you would end up with multiple configuration files to maintain. Alternatively, if there is a web server in your office, you could move the configuration file to it and set `FOO_CONFIG` on all the computers to retrieve this configuration file via `curl`.[2]

```
FOO_CONFIG="exec#curl -sS http://host/path/to/foo.cfg"
```

Recall from Section 2.11 on page 14 that the *adaptable configuration* features in Config4* enable a configuration file to adapt its contents for different computers, operating systems or users. Because of this, a single configuration file stored, say, on a centralized web server can be used for all users of the Foo application within your organization.

[2] The `curl` utility was discussed in Section 3.1 on page 20.

### 3.6.2 Parsing Embedded Configuration

There is an overloaded version of `parse()` that takes two parameters. You can use this two-parameter version to parse configuration information that is stored in a string, as this C++ example illustrates.

```
const char * str = "message = \"Hello, World\";";
cfg->parse(Configuration::INPUT_STRING, str);
```

Constructing a configuration string manually is tedious for two reasons. First, as the above example illustrates, you have to escape double quotes with a backslash. Second, compilers place limits on the maximum length of string literals; if you wanted, say, a 50KB configuration string, then you would have to construct this by concatenating numerous smaller strings.

The `config2cpp` and `config2j` command-line utilities (discussed in Chapter 6) read an input configuration file and generate a C++ or Java class that stores the contents of the configuration file in an instance variable. The generated class automates the tedious escaping of double quotes and concatenating short string literals to produce a monolithic configuration string. You can access this configuration string by invoking the public `getString()` operation on the generated class.

The `config2cpp` and `config2j` utilities make it easy to generate a configuration string that can be embedded in an application. This can be useful in an embedded system that does not contain a file system.

Version 1.2 of Config4J introduces a way to specify that the desired configuration file exists on the classpath.

```
cfg.parse(Configuration.INPUT_CLASSPATH, "path/to/file.cfg");
```

This can be specified in a more convenient form, as shown below.

```
cfg.parse("classpath#path/to/file.cfg");
```

This Java-specific capability is useful because it makes it possible to store an embedded configuration file as a resource file within a JAR file for an application. Doing this is more convenient than using `config2j` to compile the configuration file into a Java file.

### 3.6.3 Using Fallback Configuration

An important use of embedded configuration strings is to enable an application to have default configuration that can be overridden by an optional configuration file specified by, say, an environment variable or

command-line argument. A primitive way to do this is shown below in Java syntax.

```
Configuration cfg = Configuration.create();
String cfgFile = cfg.getenv("FOO_CONFIG");
if (cfgFile != null) {
    cfg.parse(cfgFile);
} else {
    cfg.parse(Configuration.INPUT_STRING,
              EmbeddedConfig.getString());
}
```

This method is primitive because it is an *either-or* approach: the configuration is obtained from *either* a file *or* an embedded string. This is acceptable if there are only a handful of configuration variables. However, if the application uses hundreds of configuration variables, then it is not convenient for a user to have to write such a large configuration file when she might want only a few configuration variables to have non-default values.

It would be preferable to allow the configuration file to contain just a few variables and for the application to automatically "fallback" to an embedded configuration string for variables not specified in the configuration file. Config4* provides support for such fallback configuration; you can see an example of its use in Figure 3.4.

Figure 3.4: Fallback configuration

```
Configuration cfg = Configuration.create();
String cfgFile = cfg.getenv("FOO_CONFIG");
if (cfgFile != null) {
    cfg.parse(cfgFile);
}
cfg.setFallbackConfiguration(Configuration.INPUT_STRING,
                             EmbeddedConfig.getString());
```

Using fallback configuration involves three steps. First, you create an empty configuration object. Second, you parse a configuration source, *if* the user has specified one. Third, you call `setFallbackConfiguration()` to apply a fallback configuration object to the main configuration object. The fallback configuration object, which contains default values for all configuration variables used by the application, is typically created by invoking the `getString()` operation on a class that was generated by `config2cpp` or `config2j`.

The semantics of fallback configuration can be understood by considering the statement below.

```
str = cfg.lookupString(scope, "log.level");
```

The `lookupString()` operation first searches in the main configuration object for the `log.level` variable in the scope specified by the `scope` parameter. If the variable is not found, then the operation searches in the *global scope* of the fallback configuration object for the `log.level` variable. The global scope is used in the fallback configuration object because a scope denotes the name of an application but fallback configuration applies to all applications.

## 3.7 Default Values

Although embedded fallback configuration is useful in applications, some people may think it is overkill if they just want to quickly hack together a short program that uses a few configuration variables. For this reason, Config4* provides a alternative mechanism, which is an extra optional parameter (denoting a default value) that can be passed to lookup operations. For example, the first Java statement below will throw an exception if the specified variable is missing from the configuration file, but the second statement will return `"/tmp"`.

```
logDir = cfg.lookupString(scope, "log.dir");
logDir = cfg.lookupString(scope, "log.dir", "/tmp");
```

## 3.8 Listing the Contents of a Scope

You can invoke `listFullyScopedNames()` to get a sorted list of the names of all entries (that is, variables and scopes) contained within a scope.

```
String[] names = cfg.listFullyScopedNames(scope, "",
                                  Configuration.CFG_SCOPE_AND_VARS, true);
```

The first two parameters to `listFullyScopedNames()` are a *scope* and *local name* within that scope. These parameters are combined to form a fully-scoped name, as discussed in Section 3.4 on page 22. In practice, you typically use an empty string for the *local name* parameter, unless you want to get a listing of a nested scope within the main scope for an application.

The third parameter is an integer bit mask that specifies what kind of entries you want to be listed. The `Configuration` class defines Java integer constants or C++ `enum` values that you can use.

- `CFG_STRING`. List the names of string variables.
- `CFG_LIST`. List the names of list variables.
- `CFG_VARIABLES`. List the names of string and list variables.
- `CFG_SCOPE`. List the names of scopes.
- `CFG_SCOPES_AND_VARS`. List all names (scopes and variables).

The final parameter indicates if `listFullyScopedNames()` should recurse into nested scopes (`true`) or just list entries in the stated scope (`false`).

At the start of this chapter, I showed a scope called `foo_srv`. The above call to `listFullyScopedNames()` for that scope returns the following list of strings.

```
foo_srv.log
foo_srv.log.dir
foo_srv.log.level
foo_srv.timeout
```

Calling the same operation but specifying `false` for the `recursive` parameter returns the following.

```
foo_srv.log
foo_srv.timeout
```

By calling the operation with a value other than `CFG_SCOPE_AND_VARS`, you can get a list of the names of just string variables (`CFG_STRING`), just list variables (`CFG_LIST`), both string and list variables (`CFG_VARIABLES`), or just scopes (`CFG_SCOPE`).

### 3.8.1 Local and Fully-scopes Names

When you call `listFullyScopedNames()`, all the strings in the returned list are *fully-scoped* names, so they have the name of the scope followed by a period (`"foo_srv."`) as a prefix. If you do not want this prefix then you call call `listLocallyScopedNames()` instead.

### 3.8.2 Determining the Type of an Entry

Once you get a list of names within a scope, you may want to iterate over the list of names and process each one by calling, say, `lookupString()` or `lookupList()`. Obviously, to know which of these operations you should call, you need to know the *type* of a named entry. You can determine this by calling `cfg.type(scope, localName)`. The value returned from this operation is one of the following integer constants.

- `CFG_STRING`. The entry is a string variable.
- `CFG_LIST`. The entry is a list variable.
- `CFG_SCOPE`. The entry is a scope.
- `CFG_NO__VALUE`. The entry does not exist.

### 3.8.3 Filtering Results with Patterns

The `listFullyScopedNames()` and `listLocallyScopedNames()` operations can take an additional `String` or `String[]` parameter that specify one or more wildcarded patterns.

```
String[] names = cfg.listLocallyScopedNames(scope, "",
                     Configuration.CFG_SCOPE_AND_VARS, true, "time*");
```

If you pass this extra parameter, then a name is included in the returned list only if the name matches at least one of the patterns. Within a pattern, `"*"` matches zero or more characters. For example, `"time*"` matches `"timeout"` but does not match `"log.dir"`.

## 3.9 Working with Uid entries

Config4* provides operations that make it easy to access entries (variables and scopes) whose names start with a `"uid-"` prefix.

### 3.9.1 Expanded and Unexpanded Names

A name like `uid-000000042-foo` is said to be in its *expanded* form, while `uid-foo` is said to be in its *unexpanded* form. You can convert a name from its expanded form into its unexpanded form with the `unexpand()` operation.

```
String name = "uid-000000042-foo.bar.uid-000000043-acme";
String unexpandedName = cfg.unexpand(name);
```

After executing the above code, the `unexpandedName` variable has the value `"uid-foo.bar.uid-acme"`.

Calling `unexpand()` on a string that does *not* contain `"uid-"` returns the same string. For example, calling `unexpand("foo.bar.acme")` returns `"foo.bar.acme"`.

Curious readers may be wondering if there is an `expand()` operation that does the conversion the opposite way. Yes, there is; `expand()` embeds a different nine-digit number whenever it encounters `"uid-"` within the name. When Config4* is parsing a file, it calls `expand()` for every name it encounters. It is unlikely you will need to call `expand()` from your own code.

### 3.9.2 The `uidEquals()` Operation

The `uidEquals()` operation takes two parameters. It calls `unexpand()` on both of its parameters and returns true if the unexpanded names are identical.

```
name = ...;
if (cfg.uidEquals("uid-foo", name)) { ... }
```

### 3.9.3 Processing Uid Entries in Sequence

The `listFullyScopedNames()` and `listLocallyScopedNames()` operations return a *sorted* list of names. This guarantees that the relative order of uid names in the list reflects the order of those entries in the input configuration file. As a concrete example, consider the configuration file in Figure 3.5 (which, for convenience, is a copy of a figure from the previous chapter).

Let us assume we want to process each `uid-recipe` scope in order and, within each of these scopes, we want to process each `uid-step` in order. You can do this with the code in Figure 3.6.

The `processRecipeFile()` operation parses a configuration file and calls `listLocallyScopedNames()` to obtain a sorted list of the `uid-recipe` scopes. Then it calls `processRecipe()` to process each of these scopes.

The body of `processRecipe()` calls `lookupString()` and `lookupList()` to get the values of the `name` and `ingredients` variables. Then it calls `listLocallyScopedNames()` to get a sorted list of the `uid-step` string variables, and uses a `for`-loop to process each of these in turn.

Figure 3.5: File of recipes

```
uid-recipe {
    name = "Tea";
    ingredients = ["1 tea bag", "cold water", "milk"];
    uid-step = "Pour cold water into the kettle";
    uid-step = "Turn on the kettle";
    uid-step = "Wait for the kettle to boil";
    uid-step = "Pour boiled water into a cup";
    uid-step = "Add tea bag to cup & leave for 3 minutes";
    uid-step = "Remove tea bag";
    uid-step = "Add a splash of milk if you want";
}
uid-recipe {
    name = "Toast";
    ingredients = ["Two slices of bread", "butter"];
    uid-step = "Place bread in a toaster and turn on";
    uid-step = "Wait for toaster to pop out the bread";
    uid-step = "Remove bread from toaster and butter it";
}
```

## 3.10 Schema Validation

A *schema* is a blueprint or definition of a system. For example, a database schema defines the layout of a database: its tables, the columns within those tables, and so on. It is common for a schema to be written in the same syntax as the system it defines. For example, a database's schema might be stored within a table of the database itself.

Another technology that uses schemas is XML. The first schema language for XML was called *document type definition* (DTD). Many people felt DTD was sufficient to define schemas for text-oriented XML documents, which tend to have a simple structure, but not flexible enough to define schemas for more structured, data-oriented XML documents. Because of this, several competing XML schema languages were defined, including XML Schema and RELAX NG.

By itself, a schema it not very useful; you also need to have a piece of software, called a *schema validator*, that can compare a system (database, XML file or whatever) against the system's schema definition and report errors. Within the Config4* library is a class called `SchemaValidator` that, as its name suggests, implements a schema validator. In this section I provide a quick overview of this schema validator; you can find full details in Chapter 9.

Figure 3.6: Code to process the file of recipes

```
void processRecipeFile()
{
    String[]       recipeNames;
    Configuration  cfg;

    cfg = Configuration.create();
    cfg.parse("recipes.cfg");
    recipeNames = cfg.listLocallyScopedNames("", "",
                            Configuration.CFG_SCOPE,
                            false, "uid-recipe");
    for (int i = 0; i < recipeNames.length; i++) {
        processRecipe(cfg, recipeNames[i]);
    }
}
void processRecipe(Configuration cfg, String scope)
{
    String[]       ingredients;
    String         name;
    String[]       stepNames;

    name = cfg.lookupString(scope, "name");
    ingredients = cfg.lookupList(scope, "ingredients");
    ... // process name and ingredients
    stepNames = cfg.listLocallyScopedNames(scope, "",
                            Configuration.CFG_STRING,
                            false, "uid-step");
    for (int i = 0; i < stepNames.length; i++) {
        step = cfg.lookupString(scope, stepNames[i]);
        ... // process step
    }
}
```

Figure 3.7 shows a scope, `foo`, in a configuration file. Figure 3.8 shows some Java code that defines a schema for the `foo` scope, parses the configuration file and then uses the `SchemaValidator` class to compare the contents of the `foo` scope against the schema.

Within Figure 3.8, the schema is defined as an array of strings (lines 5–12). Within this schema definition, ignore the fist two lines (strings starting with `"@typedef"`) for the moment. The next line defines an entry called `idle_timeout` of type `durationMilliseconds`. You can see that this describes the `idle_timeout` variable within the `foo` scope in

Figure 3.7: A configuration file to be validated

```
foo {
    idle_timeout = "2 minutes";
    log_level = "3";
    log_file = "/tmp/foo.log";
    price_list = [
        #  item      colour     price
        #---------------------------------
          "shirt",  "green",  "EUR 19.99",
          "jeans",  "blue",   "USD 39.99"
    ];
};
```

Figure 3.8: Code that performs schema validation

```
 1  import org.config4j.*;
 2  ...
 3  String scope = "foo";
 4  SchemaValidator sv = new SchemaValidator();
 5  String schema[] = new String[] {
 6      "@typedef colour = enum[red, green, blue]",
 7      "@typedef money = units_with_float[EUR, GBP, YEN, USD]",
 8      "idle_timeout = durationMilliseconds",
 9      "log_level = int[0, 5]",
10      "log_file = string",
11      "price_list = table[string,item, colour,colour, money,price]"
12  };
13  Configuration cfg = Configuration.create();
14  try {
15      cfg.parse(cfg.getenv("FOO_CONFIG"));
16      sv.parseSchema(schema);
17      sv.validate(cfg, scope, "");
18  } catch(ConfigurationException ex) {
19      System.out.println(ex.getMessage());
20  }
```

Figure 3.7. The next line defines an entry called `log_level` which is an integer in the range `0` to `5`. The line after that defines `log_file` to be of type `string`. The types used so far (`durationMilliseconds`, `int` and `string`) are built-in types for the schema validator, so the definitions for the first three entries are straightforward.

The definition for `price_list` is more interesting. You can see from

Figure 3.7 that this variable is a list of string, but the list is formatted to look like a table with three columns. The schema definition defines this entry to be a `table` in which the first column is called `item` and is of type `string`, the second column is called `colour` and is of type `colour`, and the last column is called `price` and is of type `money`. The types `colour` and `money` are *not* built-in types for the schema validator. Instead, the lines in the schema starting with `"@typedef"` define these types. You can see that `colour` is defined to be an enum of three possible values (`red`, `green` or `blue`). The `money` type is defined to be a string of the form `"<units> <float>"` where the `<units>` can be one of: `"EUR" "GBP" "YEN"` or `"USD"`.

After the configuration file has been parsed (line 15), the code uses a `SchemaValidator` object to parse the `schema` parameter and stores it in a more efficient format (line 16). Then the `validate()` operation (line 17) is used to validate the specified `scope` of the specified configuration object against the schema. If `validate()` encounters an error, then it reports the error by throwing an exception. The `catch` clause (lines 18–20) prints out the text of the exception.

### 3.10.1 Informative Error Messages

You can see from Figure 3.8 that the schema language (lines 5–13) is very compact and easy to understand, and that the API for using the schema validator (lines 17–18) is equally compact and easy to use. You may be wondering: if there are any errors in the configuration file, does the schema validator report easy-to-understand error messages? The answer is yes, as the following examples illustrate.

If you misspell `log_level` as `logLevel` then the schema validator reports the following error.

```
foo.cfg: the 'foo.logLevel' variable is unknown
```

If `log_level` is set to `"255"` then the schema validator reports the following error.

```
foo.cfg: bad int value ('255') for 'foo.log_level':
outside the permitted range [0, 5]
```

If `"car"` appears instead of `"green"` in the `colour` column of `price_list` then the schema validator reports the following error.

```
foo.cfg: bad colour value ('car') for the 'colour' column
in row 1 of the 'foo.price_list' table: should be one of:
'red', 'green', 'blue'
```

### 3.10.2 Schemas for Uid Entries

If you want to define a schema for a file that contains uid entries, then you specify the unexpanded form of uid names. For example, Figure 3.9 shows a schema for the recipes file in Figure 3.5 on page 31.

Figure 3.9: Schema validation for the recipes file

```
String schema[] = new String[] {
    "uid-recipe = scope",
    "uid-recipe.name = string",
    "uid-recipe.ingredients = list[string]",
    "uid-recipe.uid-step = string"
};
```

## 3.11 Summary

The API of Config4* is simple. As demonstrated in Figures 3.1 and 3.2 on page 20, a basic application needs just three steps to use Config4*: (1) create an empty `Configuration` object; (2) `parse()` a configuration file; and (3) call `lookup<Type>()` operations to access configuration variables in a type-safe manner. Doing that will enable the application to avail of some important benefits of Config4*, such as *adaptable* and *centralised* configuration.

Other features of Config4* can be accessed with a few extra operation calls.

- *Integration with command-line options.* Figure 3.3 on page 23 illustrates how an application can use `"-set name value"` command-line options to insert name-value pairs into a `Configuration` object *before* parsing a configuration file. This trivial step makes it feasible for a configuration file to use the conditional assignment operator (`"?="`), so that command-line options can override default values in a configuration file.

- *Fallback configuration.* As discussed in Section 3.6.3 on page 25, the `config2cpp` and `config2j` utilities can convert a configuration file into a (class wrapper around a) string that can be embedded inside an application. This embedded string can be used to populate a *fallback* `Configuration` object that is then attached to the application's *main* `Configuration` object. Once this has been done,

a call to a `lookup<Type>()` operation on the *main* `Configuration` object will first search for the value in the main object; if it is not present there then the operation will return the value from the fallback object. In this way, an application can be highly configurable yet can work with just a minimal (or even no) external configuration file.

- *Schema validation.* As discussed in Section 3.10 on page 31, Config4* provides an easy-to-use schema validator that can perform useful checks on the contents of a configuration file. For example, it can report variables or scopes with misspelt or unknown names. The schema validator also makes it possible for a list of strings to be formatted, and interpreted, as a table with type-specific columns.

- *Uid entries.* The `"uid-"` prefix on the names of variables and scopes provides an elegant way for a Config4* file to store ordered lists of items. As discussed in Section 3.9 on page 29, the Config4* API provides operations that make it easy for an application to process `"uid-"` entries.

These capabilities of Config4* are very powerful and flexible, yet the Config4* API remains extremely easy to use.

# Chapter 4

# Comparison with Other Technologies

## 4.1 Introduction

Config4* is not the only option you have for making applications configurable. Many other choices exist, including command-line arguments, environment variables, writing your own configuration-file parser, using Java properties files, or using XML files. In this chapter, I compare these alternative approaches to Config4*.

## 4.2 Command-line Options & Environment Variables

When you start writing an application, you might think the application needs only a small amount of configuration information and conclude that this need can be met by the use of command-line options. Over time, the amount of configuration information used by an application tends to grow. If the number of command-line options grows to more than, say, 10, then users are likely to think the application is difficult to configure. In contrast, a configuration file containing potentially hundreds of entries can be easy to use and maintain, especially if fallback configuration is used to provide useful default configuration values.

The use of environment variables to store configuration information

shares a limitation with the use of command-line options: if an application uses more than a handful of environment variables, then most users will think the application is difficult to configure.

Environment variables suffer from another drawback, which is that for a time it was difficult for a Java-based application to access environment variables. The first version of Java provided `System.getenv()` for accessing environment variables. However, later on Sun realized that some operating systems, such as MacOS, did not support environment variables. Sun was promoting Java as a highly portable language, and had even coined a slogan to reflect this: *Write Once, Run Anywhere.* Sun decided it did not make sense to encourage people to rely on a not-universally-supported concept such as environment variables. Because of this, Sun deprecated the use of `System.getenv()` and encouraged developers to use Java system properties as a replacement. A Java system property can be set by supplying a command-line option of the form `-D<name>=<value>` to the Java interpreter. Several years later, Apple decided to re-implement the Macintosh operating system on top of a UNIX base; doing so introduced environment variables to the Macintosh, but there may still be other operating systems that do not support environment variables. Java 1.5 has *undeprecated* `system.getenv()` and this method now returns `null` if the environment variable does not exist or if the operating system does not support environment variables.

As a side note, developers using a pre-1.5 version of Java can access environment variables, albeit in an indirect manner. This is done by calling `java.lang.Runtime.getRuntime().exec()` to execute an external command (for example, `"cmd /c set"` on Windows or `"env"` on UNIX) that prints *name=value* pairs for all environment variables. You parse the output of the executed command to gain read-only access to the environment variables. The Ant utility (http://ant.apache.org) uses this technique so that environment variables can be used in its build files. Config4J uses the same technique to enable access to environment variables from configuration files.

## 4.3 Writing Your Own Configuration Parser

Many developers think to themselves "It's not *that* difficult to write a configuration-file parser; I could hack together something in a few hours and with just a few hundred lines of code". However, a configuration-file parser built in such a short period of time is likely to offer very lim-

ited functionality. For example, perhaps the parser accepts *name=value* statements but the value must be a string literal that is terminated by the end of line. This means that none of the following are supported: (1) long values that extend over several lines, (2) values that are *lists*, (3) values defined in terms of environment variables or previously defined configuration variables, (4) scopes for grouping related *name=value* statements, (5) if-then-else statements that enable a file to encapsulate configuration information that varies between users or machines, (6) the ability for one configuration file to include other files, or (7) the ability to obtain configuration from, say, a website or a database.

Another problem with writing a configuration-file parser is that it is not sufficient to just write the code. You also need to test and document it; ideally, you should write both a user guide and programming guide. Writing tests and documentation might not be particularly difficult, but these tasks require time. An initial plan to "hack together something in a few hours and with just a few hundred lines of code" can easily turn into a multi-week job. And the likely result is that you still have a configuration-file parser with the numerous limitations listed in the previous paragraph.

A final problem with this approach is that development teams for countless projects have written their own configuration-file parsers. Each development team's parser tends to have slight differences in the syntax it accepts. The result is that users have to learn a different configuration syntax for each application they use.

A better approach is for development teams, globally, to standardize on using the *same* configuration-file syntax in all their applications. Of course, if one configuration-file syntax is to be used by many applications, then the syntax needs to be very flexible, parsers for that syntax need to be available in different languages, there should be good user- and programmer-oriented documentation available, and the parsers should be available under a license that does not hinder their use in open-source or proprietary projects. These are precisely the goals of Config4*.

## 4.4 Java Properties Files

The standard Java class library provides a configuration-file parser, although Java uses the term *properties* file rather than *configuration* file. The syntax rules of a Java properties file are as follows.

Lines starting with `#` or `!` denote comments.

The backslash character ("\") is used as an escape mechanism. For example, "\t" and "\n" denote tab and newline characters. A backslash at the end of a line denotes line continuation, thus enabling a long string to be split across several lines. The input file is assumed to be in the ISO-Latin-1 encoding (ISO-Latin-1 is an 8-bit character set that contains the characters of US-ASCII plus accented characters for some European languages). Internally, Java stores strings in a Unicode format, so each ISO-Latin-1 character is converted to its Unicode equivalent while the properties file is being parsed. The escape sequence \uxxxx, where each x denotes a hexadecimal digit, can be used to represent an arbitrary Unicode character by its hexadecimal code point.

Entries in a Java properties file consist of *name-value* pairs. The name and the value can be separated by "=" or ":", with optional whitespace surrounding the separator. Alternatively, the name and value can be separated by just whitespace, that is, without "=" or ":". If a name but no value is specified, then an empty string is used as the value.

There are several significant problems with Java properties, as I discuss in the following subsections.

### 4.4.1 Unwanted Whitespace at the End of a Line

Consider the following line from a properties file.

```
logFile = /tmp/foo.log
```

That line *appears* to set logFile to the value "/tmp/foo.log". However, if there are any spaces at the end of the line, then those spaces become part of the value, which is almost certainly *not* what is desired. Trailing whitespace is a common cause of misconfigured Java applications, and it can be difficult to diagnose such misconfiguration because the whitespace is invisible in most text editors. In Config4*, strings are enclosed within double quotes which prevents this cause of misconfiguration.

### 4.4.2 Lack of Syntax Checking

Figure 4.1 shows an example of a Java properties file. The first two lines use "=" and ":" to separate the name and value. The third and fourth lines use just a space to separate the name and value. For example, in line 3, the name is Java and its value is "properties accepts this text". The last line of the properties file contains just a name (&*%!)[1]

[1] &*%! is not really a name. It's more of a curse at the lack of error checking in a Java properties file.

so an empty string is used as its value.

Figure 4.1: Example of input acceptable to Java properties

```
Roses=red
Violets : better
Java properties accept this text
Without raising an error
&*%!
```

The syntactic flexibility permitted in a Java properties file is problematic because it means that very little syntax checking is performed. In fact, almost any file — whether it be a text file or a binary file — is acceptable as input to the Java properties parser. The *only* syntactic error that might be raised is if the file contains the characters `"\u"` and these are *not* followed by four hexadecimal digits.

One principle of good programming is *fail fast* [Ray03], which means that if a program is going to fail, then it should fail at the earliest opportunity because this makes it easier for users to diagnose the problem. The Java properties file parser ignores the fail-fast principle, instead accepting almost any garbage as input and letting other parts of an application report an error when they find that an expected property has not been set.

### 4.4.3 Semantically Poor

The names in the *name=value* pairs of a properties file are in a flat namespace. The names can have embedded periods (`"."`) but that seems to be a side-effect of allowing arbitrary characters (such as `&*%!`) in names rather than an intent to support hierarchical names. Certainly, there is nothing in the Java properties API that supports the concept of hierarchical names. When you have only a flat namespace in a configuration file, then you typically end up having a separate configuration file for each application. In contrast, Config4* provides explicit support for hierarchical scopes through its syntax and API; this makes it feasible to store configuration information for several (usually related) applications in a single file.

Another limitation of properties files is that there is no support for values that are *lists* of strings. It is common for developers to work around this limitation by writing code that splits a property value into a list based on occurrences of, say, a comma (`","`). However, having to

do this is tedious. The approach taken in Config4* of providing explicit support for lists is better.

Other useful capabilities of Config4* have no counterpart in Java properties files. For example, there is no ability to define a variable in terms of other variables, and there is no support for reusing configuration information, such as the `@include` and `@copyFrom` statements in Config4*. Also properties files have no support for accessing centralized configuration information by executing an external command, such as `"exec#..."` in Config4*, and no support for *adaptive configuration* (Section 2.11 on page 14).

### 4.4.4 Type-unsafe Lookup API

The API of the `Properties` class enables developers to access property values only as strings. It is common for a property to be a non-string type, such as an integer, floating-point number, boolean or list. Whenever type-safe access to a property value is required, developers must write code that retrieves the property value as a string, converts it to another data type, and throws a suitable exception if the data-type conversion fails. Code for doing this is not difficult to write, but it is tedious and the need to write such code is common enough that it would have been better for the `Properties` class to provide this functionality. Config4* provides such type-safe lookup operations.

## 4.5 Platform-specific Configuration Files

It is common for a software company to write a configuration-file parser suitable for the needs of its own products. Having done this, the company may then decide to expose the API of the configuration-file parser so that their customers can use it too. Two famous examples come from Microsoft. Microsoft wrote a parser for its `".ini"` files that were used in Windows 3.1, and exposed the API of this parser so companies that wrote applications for Windows could make use of it. Then when Microsoft released Windows 95, it switched from `".ini"` files to the Windows Registry. Again, it exposed the API for this and many companies made use of it in their products. Another famous example is X11, a windowing system widely used on UNIX machines. X11 exposes an API for parsing its configuration files, called *X resource files*. Countless other examples can be found in companies that sell framework libraries.

If you are writing a Windows-based application, then it is easier to use the Windows-supplied API for accessing the registry rather than write your own configuration-file parser. Likewise, if you are writing an X11-based application, then it is easier to retrieve your application-specific configuration information from an X11 resource file rather than write your own configuration-file parser. And if you are writing an application that uses a framework library that happens to provide a configuration-file parser, then... Well, you get the idea.

If your application runs on only one *platform* (operating system, windowing system, framework library and so on), then making use of that platform's configuration-file parser seems like a great idea. However, it is common for an application to be developed *initially* for one platform and later be ported to other platforms. As soon as that happens, your use of a platform-specific configuration-file parser becomes a liability. Countless software teams have run into this problem over the years. If you use an Internet search engine, then you will find evidence of different groups who have implemented their own parser for, say, `".ini"` files because they have ported a Windows-specific application to another operating system. Likewise, some groups have implemented a Java properties file parser in other languages because their initially Java-only project grew to include components written in other languages.

I am not going to critique numerous platform-specific, configuration-file parsers. Aside from me never having seen one that has as much flexibility as Config4*, they all suffer from the same fundamental limitation, which is that they are platform specific.

## 4.6 XML-based Configuration Files

XML parsers are available for a wide variety of programming languages and operating systems. This means that XML provides a platform-neutral configuration file format. The platform-neutral characteristic of XML is probably a significant reason why more and more developers are using XML for storing configuration information. However, there are some significant drawbacks to using XML to store configuration information, as I now discuss.

### 4.6.1 Verbosity

XML can be easy to read when it is used for *infrequent* markup in a text-oriented document. However, when XML is used for structured data,

then the amount of syntactic baggage imposed by the start tags and end tags becomes visually distracting. You can see this by comparing the Config4* file in Figure 4.2 with similar information expressed in XML format in Figure 4.3.

Figure 4.2: Example Config4* document

```
fooSrv {
    timeout = "2 minutes";
    log {
        dir = "C:\foo\logs";
        level = "0";
    }
}
```

Figure 4.3: Example XML document

```
<fooSrv>
    <timeout>2 minutes</timeout>
    <log>
        <dir>C:\foo\logs</dir>
        <level>0</level>
    </log>
</fooSrv>
```

Some people may think the XML would be more compact if it was written to make use of attributes, but a quick look at Figure 4.4 shows that this is not necessarily the case.

Figure 4.4: Example XML document using attributes

```
<scope name="fooSrv">
    <property name="timeout" value="2 minutes"/>
    <scope name="log">
        <property name="dir" value="C:\foo\logs"/>
        <property name="level" value="0"/>
    </scope>
</scope>
```

Many developers who work with structured XML files on a daily basis claim “you eventually get used to the verbosity”. And, of course,

even if some developers do *not* get used to it, they may still endure it simply because they are paid to do what their employers tell them to do. However, if end users dislike XML's verbosity, then they may decide that an application with an XML-based configuration file is too difficult to use, and so go in search of simpler-to-use application from a competing vendor.

### 4.6.2 Limited Functionality

Once you look beyond the syntactic differences of XML and Config4*, it is possible to compare the functionality they offer.

Several pieces of functionality are similar. First, the ability to nest elements in an XML file provides functionality similar to the nested scopes of Config4*. Second, in Config4*, a value can be either a string or a list, while in XML, a value can be a string, and it is possible to use nested elements to denote a list. Third, an XML document may contain several identically-named elements. Config4* provides a similar capability through the `"uid-"` prefix on identifiers (Section 2.12). Finally, an application can iterate over the elements in an XML document in the order in which they appeared in the source document. In contrast, Config4* stores parsed information in hash tables; these provide fast lookup times but they do not preserve the original order in which the entries appeared within the source file. However, if processing entries in order is important, then this can be achieved through use of the `"uid-"` prefix on the names of entries.

Several capabilities in Config4* have no counterpart in XML. These are: (1) statements, such as `@include` or `@copyFrom`, that enable configuration to be reused; (2) the ability to define one configuration variable in terms of other configuration variables; or (3) *adaptive configuration* (Section 2.11 on page 14). None of these capabilities is provided by an out-of-the-box XML parser. Instead, countless XML-based projects have had to implement such functionality by parsing an XML document and then doing some post-processing of the generated DOM tree.

### 4.6.3 Checking the Correctness of Input Files

Writing code to check the validity of an XML file is very tedious. To avoid this, some developers use XML Schema to define what can legally appear in an XML-based configuration file and then parse configuration files with a schema-validating XML parser. However, this approach has

several drawbacks. First, XML Schema has a steep learning curve.[2] Second, schema files are extremely verbose. Third, it can be quite difficult to understand some of the error messages produced by schema validators.

In contrast, the Config4* schema validator has a trivial learning curve (it is described completely in 12 pages in Chapter 9), its schema definitions are very compact, and its error messages are easy to understand.

### 4.6.4 Memory Usage

Use of Config4* adds a few hundred KB to an application's executable size, and the amount of RAM consumed when a configuration file is parsed is quite small: about 2.5 times the size of the configuration file. In contrast, the library for a schema-validating XML parser can add several megabytes to an application. In addition, the DOM tree built by most parsers can also consume a lot of memory. Many functionally-rich applications may not care about a few megabytes of overhead for parsing XML-based configuration files. However, such overhead is unacceptable for many smaller applications.

## 4.7 A Critique of Config4*

So far in this chapter I have compared Config4* with some other configuration technologies and, in so doing, have pointed out what I feel are the drawbacks of these other technologies. I now turn my focus on Config4* to discuss its drawbacks.

One significant drawback of Config4* is that, currently, there are implementations for only two languages: C++ and Java. I hope implementations for other languages will follow in time.

A second significant drawback of Config4* is that its internationalization and localisation support is a work in progress (see the *Config4* Maintenance Guide* for a discussion of these issues). I hope that, over time, these deficiencies will be addressed with the support of the open-source community.

These drawbacks are due to the young age of Config4*, so a bit of maturing should resolve both drawbacks. In contrast, the other configuration technologies discussed in this chapter have been around for over

[2] If you wish to learn XML Schema, then I recommend *Definitive XML Schema* by Priscilla Walmsley [Wal02]. The book is excellent, but the fact it is about 500 pages long indicates that XML Schema is complex and has a steep learning curve.

a decade. The drawbacks of those other technologies are due to inherent design limitations rather than immaturity.

## 4.8 Summary

In this chapter I have compared Config4* with alternative technologies for accessing configuration information. The purpose of the comparisons is not to ridicule or demonise any of the alternative approaches to providing configuration information. Rather, my purpose is to show that, when it comes to configuration parsers, there is a widespread underappreciation for the importance of issues such as: (1) adaptable configuration, (2) centralizable configuration, (3) hierarchical scopes, (4) typesafe access, (5) ease of use for developers, (6) ease of use for end users, and (7) portability across operating systems and languages. Most existing configuration technologies score poorly against most of these criteria. Config4* is useful in its own right, but by scoring high in these seven criteria, it raises the bar for other (present and future) configuration technologies.

I am writing this paragraph in 2011 and, as far as I know, Config4* is by far the best configuration parser in the world. I will be disappointed if I can still make that claim in, say, 5 years time. By highlighting the ways in which Config4* is superior, I am laying down a challenge to the developers of other configuration technologies. *Please*, take the best ideas from Config4* and innovate to produce something better. Configuration parsers have been depressingly mediocre for decades. It is time for us to make them better. Significantly better.

# Part II

# Infrastructure

# Chapter 5

# Config4* Security

## 5.1 The Need for Security

There are three ways that Config4* can execute an external command. First, an external command can be specified when parsing a configuration source.

```
cfg.parse("exec#command");
```

Second, within a configuration file, you can `@include` the output of executing an external command.

```
@include "exec#command";
```

Finally, within a configuration file you can use `exec()` to execute an external command.

```
name = exec("command");
```

Each of these three cases presents the same security risk: if Config4* permits *arbitrary* external commands to be executed, then somebody could arrange for a malicious command to be executed. For example, imagine the damage that could be caused by somebody adding the following to a configuration file.

```
@if (osType() == "windows") {
    @include "exec#del /F /S /Q C:\";
} @elseIf (osType() == "unix") {
    @include "exec#rm -rf /";
}
```

We need a way to prevent such malicious commands from being executed, while allowing non-malicious commands, such as `curl` and `hostname`, to be executed.

## 5.2 The Config4* Security Mechanism

The security mechanism in Config4* is driven by three configuration variables: `allow_patterns`, `deny_patterns` and `trusted_directories`. An attempt to execute a command will succeed only if *all* of the following conditions are true.

- The command matches one or more patterns in `allow_patterns`.
- The command does *not* match any patterns in `deny_patterns`.
- The first word of the command is a file that exists in one of the directories in `trusted_directories`. For example, if the command line is `"curl -sS http://host/file.cfg"` then `curl` must exist on one of the directories in `trusted_directories`. On Windows, Config4* will check for the existence of the file with no extension, a `".exe"` extension and a `".bat"` extension.

When pattern matching is being performed, `"*"` is treated as a wildcard that can match zero or more characters. For example, the pattern `"curl *"` matches `"curl -sS http://host/file.cfg"`. The pattern `"curl*"` (no space before `"*"`) matches the same string but it matches `"curlfoobar"` too.

If Config4* allows a command to be executed then it rewrites the command slightly to put in the full path to the executable. For example, if `curl` resides in `/usr/local/bin` (and let us assume that directory is listed in `trusted_directories`) then `/usr/local/bin/` is prefixed onto the command `"curl -sS http://host/file.cfg"` when it is being executed. This is to ensure that the executed command is one in a trusted directory rather than one in another directory that appears in `PATH`.

## 5.3 The Default Security Policy

Figure 5.1 shows the default security policy of Config4*.

The `@if`-then-`@else` statement in the default security policy makes it possible to tailor the security for different operating systems. You can see

Figure 5.1: Default security configuration

```
@if (osType() == "unix") {
    allow_patterns = ["curl *", "hostname", "uname",
                      "uname *", "ifconfig"];
    deny_patterns = ["*‘*", "*|*", "*>*"];
    trusted_directories = ["/bin", "/usr/bin", "/sbin"
                           "/usr/local/bin", "/usr/sbin"];
} @elseIf (osType() == "windows") {
    allow_patterns = ["curl *", "hostname", "uname",
                      "uname *", "ipconfig"];
    deny_patterns = ["*‘*", "*|*", "*>*"];
    trusted_directories = [
        getenv("SYSTEMROOT") + "\system32"
    ];
} @else {
    allow_patterns = [];
    deny_patterns = ["*"];
    trusted_directories = [];
};
```

from Figure 5.1 that the security policy varies slightly between UNIX and Windows; for other (unknown) operating systems, the security policy denies all attempts to execute commands. Over time, it is likely that Config4* will be ported to other operating systems, and the default security policy will be updated to reflect this.

On UNIX, `allow_patterns` permits the use of `curl`, `hostname` (without any command-line arguments), `uname` (with and without command-line arguments), and `ifconfig` (without any command-line arguments). The `deny_patterns` denies the use of back quotes, pipes and redirection of output. The use of back quotes and pipes is prohibited because they provide ways to combine potentially malicious commands with benign commands, as the examples below show.

```
@include "exec#curl ‘rm -rf /‘";
@include "exec#curl | rm -rf /";
```

Redirection of output is prohibited for two reasons. First, Config4* needs to have access to the standard output and standard error of executed commands so there is no valid reason for a user to want to redirect standard output or standard error. Second, if redirection of output was allowed then a malicious person might use this to damage important files in the operating system. For example, a malicious person might trick somebody with root privileges to parse a configuration file containing

the following.

```
@include "exec#curl > /etc/passwd";
```

The Config4* security policy on Windows is very similar to that on UNIX. Partly this is due to Windows and UNIX having some similarities, and partly due to collections of UNIX-like utilities, such as Cygwin (www.cygwin.com), being available for Windows machines. The main difference between the default security policies on Windows and UNIX is `trusted_directories`, as can be seen in Figure 5.1.

## 5.4 Overriding the Default Security Policy

When a `Configuration` object is created, it uses the default security policy shown in Figure 5.1 on page 53. You can override this default security policy by calling `setSecurityConfiguration()`. The first (and possibly only) parameter to this operation is a configuration object that defines `allow_patterns`, `deny_patterns` and `trusted_directories`. By default, Config4* assumes that these variables are defined in the global scope. However, if the variables are defined in a nested scope then you should pass a second parameter that indicates the name of this scope. For example, consider a security policy defined in a scope called `security`.

```
security {
    allow_patterns = [ ... ];
    deny_patterns = [ ... ];
    trusted_directories = [ ... ];
}
```

The code below shows (in Java syntax) how you can set that security policy on a `Configuration` object called `cfg` prior to parsing an application configuration file.

```
Configuration cfg    = Configuration.create();
Configuration secCfg = Configuration.create();
secCfg.parse(...);
cfg.setSecurityConfiguration(secCfg, "security");
cfg.parse(...); // parse application configuration
```

That code is a bit verbose, but you can shorten it somewhat by using one of the overloaded versions of `setSecurityConfiguration()`. For example, if the security policy is defined in a file called `security.cfg` then you can use the following code.

```
Configuration cfg = Configuration.create();
cfg.setSecurityConfiguration("security.cfg", "security");
cfg.parse(...); // parse application configuration
```

If the security policy is, say, on a web server then you can use `"exec#..."` in place of the file name (as long as the default security policy in place permits that command to execute). If the security policy is in an embedded configuration string (perhaps generated with the aid of `config2cpp` or `config2j`) then you can use code like that shown below.

```
Configuration cfg = Configuration.create();
cfg.setSecurityConfiguration(
        Configuration.INPUT_STRING,
        EmbeddedSecurityConfig.getString(),
        "security");
cfg.parse(...); // parse application configuration
```

## 5.5 Summary

There are three ways in which Config4* can execute a shell command.

```
cfg.parse("exec#command");  // C++ or Java code.
@include "exec#command";     // Inside a configuration file.
name = exec("command");     // Inside a configuration file.
```

Each case presents the same security issue: we need to permit useful commands to be executed, while preventing the execution of harmful commands. Config4* does this by attaching a security policy to each `Configuration` object. A default security policy is embedded inside the Config4* library, and an application programmer can override this for a `Configuration` object by invoking the `setSecurityConfiguration()` operation.

# Chapter 6

# The `config2cpp` and `config2j` Utilities

## 6.1 Introduction

The `config2cpp` and `config2j` utilities read a configuration file and generate a C++ or Java file that contains a class wrapper around a snapshot of the file's contents. These utilities make it easy to generate a configuration string that can be embedded in an application. This can be useful in an embedded system that does not contain a file system. It is also useful if you want an application to have "fallback" configuration (discussed in Section 3.6.3 on page 25).

## 6.2 Basic Operation

The `config2cpp` utility is a compiled application, while `config2j` is a Windows batch file or UNIX shell script that executes the `main()` operation of the `org.config4j.Config2J` class.

Figure 6.1 shows a file, `Fallback.cfg`, that contains the configuration information we want to embed as the fallback configuration for an application. We can do this for a C++ or Java application by running one of the following commands:

```
config2cpp -cfg Fallback.cfg -class FallbackConfig
config2j   -cfg Fallback.cfg -class FallbackConfig
```

Figure 6.1: The file `Fallback.cfg`

```
timeout = "infinite";
log {
  dir = ".";  # current working directory
  level = "1";
}
TCP {
  buffer_size = "8 KB";
  threading_policy = "thread_pool";
  max_threads = "10";
}
SSL {
  buffer_size = "8 KB";
  threading_policy = "thread_pool";
  max_threads = "10";
}
```

Figure 6.2 shows the Java class generated from the file shown in Figure 6.1; a generated C++ class would be structurally similar. The constructor initialises two instances variables, `schema` and `str`, that can be accessed by calling the public operations `getSchema()` and `getString()`.

For the moment, ignore the initialisation of `schema` (I will discuss that in Section 6.4 on page 60), and instead look at the initialisation of `str`. That variable is initialised to hold a copy of the contents of the file given as a command-line argument to `config2j` or `config2cpp`.

By default, your application would have to create an instance of the generated class before invoking `getSchema()` or `getString()`. That explicit creation step can be avoided by using the `-singleton` command-line option, which causes the generated class to provide an automatically-created singleton object; `getString()` and `getSchema()` become `static` operations that delegate to the singleton object.

```
config2cpp -cfg Fallback.cfg -class FallbackConfig -singleton
config2j   -cfg Fallback.cfg -class FallbackConfig -singleton
```

By default, the generated class is not in any Java package or C++ namespace. The `-package <name>` or `-namespace <name>` command-line option can be used to generate the class in a specified package or namespace. For example:[1]

```
config2cpp -cfg Fallback.cfg -class FallbackConfig -singleton \
```

[1] The backslash indicates line continuation.

Figure 6.2: Java class generated by `config2j`

```
class FallbackConfig
{
  public FallbackConfig()
  {
    schema = new String[12];
    schema[0] = "SSL = scope";
    schema[1] = "SSL.buffer_size = memorySizeBytes";
    schema[2] = "SSL.max_threads = int";
    schema[3] = "SSL.threading_policy = string";
    schema[4] = "TCP = scope";
    schema[5] = "TCP.buffer_size = memorySizeBytes";
    schema[6] = "TCP.max_threads = int";
    schema[7] = "TCP.threading_policy = string";
    schema[8] = "log = scope";
    schema[9] = "log.dir = string";
    schema[10] = "log.level = int";
    schema[11] = "timeout = durationSeconds";
    str = new StringBuffer();
    str.append("timeout = \"infinite\";" + CR);
    str.append("log {" + CR);
    str.append("  dir = \".\";  # current working directory" + CR);
    str.append("  level = \"1\";" + CR);
    str.append("}" + CR);
    str.append("TCP {" + CR);
    str.append("  buffer_size = \"8 KB\";" + CR);
    str.append("  threading_policy = \"thread_pool\";" + CR);
    str.append("  max_threads = \"10\";" + CR);
    str.append("}" + CR);
    str.append("SSL {" + CR);
    str.append("  buffer_size = \"8 KB\";" + CR);
    str.append("  threading_policy = \"thread_pool\";" + CR);
    str.append("  max_threads = \"10\";" + CR);
    str.append("}" + CR);
    str.append("");
  }
  public String[] getSchema() { return schema; }
  public String   getString() { return str.toString(); }
  private String[] schema;
  private StringBuffer str;
  private static final String CR= System.getProperty("line.separator");
}
```

```
         -namespace x::y::z
config2j   -cfg Fallback.cfg -class FallbackConfig -singleton \
         -package com.example.foo
```

## 6.3 Using the Generated Class

Let's assume we have converted the file `Fallback.cfg` into a Java class called `FallbackConfig` with the following command:

```
config2j -cfg Fallback.cfg -class FallbackConfig -singleton \
         -package com.example.foo
```

Figure 6.3 shows how the `FallbackConfig` class can be used to provide fallback configuration for an application.

Figure 6.3: Example of Using Config4J

```
Configuration cfg = Configuration.create();
String cfgFile = ...
try {
  if (cfgFile != null) { cfg.parse(cfgFile); }
  cfg.setFallbackConfiguration(Configuration.INPUT_STRING,
                               FallbackConfig.getString());
} catch(ConfigurationException ex) {
  System.out.println(ex.getMessage());
}
```

The code in Figure 6.3 is straightforward. It parses a configuration file if one was specified by, say, a command-line option or an environment variable. Afterwards, it calls `setFallbackConfiguration()` to set fallback configuration based on the string in the `FallbackConfig` class. Because we had run `config2j` with the `-singleton` option, the code can call `FallbackConfig.getString()`. If we had not used the `-singleton` option, then the code would have had to explicitly create an instance of the `FallbackConfig` class, for example:

```
cfg.setFallbackConfiguration(Configuration.INPUT_STRING,
                             new FallbackConfig().getString());
```

## 6.4 Tweaking the Generated Schema

If you look again at Figure 6.2 on page 59, you will see that the class generated by `config2cpp` or `config2j` provides not just `getString()` but

also `getSchema()`. The code at the start of the constructor initializes the schema returned by `getSchema()`.

The `config2cpp` and `config2j` utilities employ the following heuristics to generate a schema from an input configuration file.

- Each scope or list variable defined in the input configuration file results in a corresponding entry in the schema.

- If an entry in the input configuration file is a string variable, then `config2cpp` and `config2j` check to see if the variable's value can be parsed as one of the built-in types: `boolean`, `int`, `float`, `durationSeconds`, and so on. If so, then its schema entry reflects that built-in type; otherwise, its schema entry indicates it is of type `string`. For example, when parsing `Fallback.cfg` (Figure 6.1 on page 58), the `timeout` variable has the value `"infinite"` so its schema entry indicates it is of type `durationSeconds`, and the `TCP.max_threads` variable has the value `"10"` so its schema entry indicates it is of type `int` (if it had the value `"10.0"` then its schema entry would indicate it to be of type `float`).

Those heuristics work well *most* of the time. However, they can make mistakes. As an example, consider the `threading_policy` variable in the `TCP` and `SSL` scopes of Figure 6.1 on page 58. That has the value `"thread_pool"`, which might be an `enum` value rather than merely a `string`. As another example, perhaps the `log.level` variable should be represented in the schema as being an integer with a *limited range* of values, such as `int[0,3]`.

You can provide `config2cpp` and `config2j` with instructions on how to generate the schema. To do this, you need to write another configuration file as shown in Figure 6.4. That file contains three variables that tweak the schema-generation heuristics of `config2cpp` and `config2j`.

The `user_types` variable defines a list of types that are to be included in the generated schema. The `wildcarded_names_and_types` variable is a three-column table. Within each row of this table, the first column contains a keyword (either `@optional` or `@required`), the second column contains the name of a configuration variables, and the third column specifies its schema type. The meaning of the `@optional` and `@required` keywords will be discussed in Section 9.2.2 on page 102; `@optional` is the appropriate keyword to use in most circumstances. The name in the second column can contain `"*"`, which acts as a wildcard that matches zero or more characters. Thus, `"*.threading_policy"` matches both

Figure 6.4: The file SchemaFineTuning.cfg

```
user_types = [
  "@typedef threadingPolicy = enum[thread_pool, thread_per_socket]"
];
wildcarded_names_and_types = [
  # optional/required   wildcarded name          Schema type
  #-------------------------------------------------------------------
   "@optional",         "log.level",             "int[0,3]",
   "@optional",         "*.threading_policy",    "threadingPolicy",
];
ignore_rules = [
];
```

"SSL.threading_policy" and "TCP.threading_policy". The ignore_rules variable is used to specify *ignore rules*, which will be discussed in Section 9.2.6 on page 109. For the example being discussed, we do not need any ignore rules, so we set this variable to be an empty list.

Having written that file, you use the -schemaOverrideCfg command-line option to pass the file to config2j or config2cpp:

```
config2j -cfg Fallback.cfg -class FallbackConfig -singleton \
         -package com.example.foo \
         -schemaOverrideCfg SchemaFineTuning.cfg
```

You can see the results in Figure 6.5.

Having generated an accurate schema, we can now use it to perform schema validation. Figure 6.6 provides an example of this.

## 6.5 Summary

The config2cpp and config2j utilities read a configuration file and generate a C++ or Java file that contains a class wrapper around a snapshot of the file's contents. These utilities make it easy to generate a configuration string that can be embedded in an application. This can be useful in an embedded system that does not contain a file system. It is also useful if you want an application to have fallback configuration.

The class generated by the config2cpp and config2j utilities can provide not just embedded configuration, but also a schema. The built-in heuristics for generating the schema definition work well most of the time. You can use the -schemaOverrideCfg command-line option to specify some tweaks for the generated schema.

Figure 6.5: The effects of tweaking the schema

```
class FallbackConfig
{
  public FallbackConfig()
  {
    schema = new String[13];
    schema[0] = "@typedef threadingPolicy = enum[thread_pool,
thread_per_socket]";
    schema[1] = "SSL = scope";
    schema[2] = "SSL.buffer_size = memorySizeBytes";
    schema[3] = "SSL.max_threads = int";
    schema[4] = "@optional SSL.threading_policy = threadingPolicy";
    schema[5] = "TCP = scope";
    schema[6] = "TCP.buffer_size = memorySizeBytes";
    schema[7] = "TCP.max_threads = int";
    schema[8] = "@optional TCP.threading_policy = threadingPolicy";
    schema[9] = "log = scope";
    schema[10] = "log.dir = string";
    schema[11] = "@optional log.level = int[0,3]";
    schema[12] = "timeout = durationSeconds";
    ... // code to initialize str omitted for brevity
  }
  ...
}
```

Figure 6.6: Using the generated schema

```
Configuration cfg = Configuration.create();
SchemaValidator sv = new SchemaValidator();
String scope = ...
String cfgFile = ...
try {
  if (cfgFile != null) { cfg.parse(cfgFile); }
  cfg.setFallbackConfiguration(Configuration.INPUT_STRING,
                               FallbackConfig.getString());
  sv.parseSchema(FallbackConfig.getSchema());
  sv.validate(cfg, scope, "");
} catch(ConfigurationException ex) {
  System.out.println(ex.getMessage());
}
```

# Chapter 7

# The **config4cpp** and **config4j** Utilities

## 7.1 Introduction

The `config4cpp` and `config4j` utilities are command-line utilities that act as wrappers for their corresponding Config4* libraries.[1] These utilities serve several purposes.

First, when you have written or edited a configuration file, you can use `config4cpp` or `config4j` to check if the file has any syntax errors or, optionally, schema validation errors.

Second, the utilities provide a way for you to "play with" the Config4* API without having to write code. As such, these utilities can shorten the learning curve for developers.

Finally, the utilities make it possible for a UNIX shell script to retrieve information from a Config4* file. This makes it possible to use Config4* to configure shell script-based applications.

### 7.1.1 Basic Operation

The `config4cpp` and `config4j` utilities work identically so, for brevity, I discuss just `config4cpp` in this chapter.

[1] The `config4cpp` utility is a compiled application, while `config4j` is a Windows batch file or UNIX shell script that executes the `main()` operation of the `org.config4j.Config4J` class.

You can obtain a usage statement by running `config4cpp` without any command-line arguments (or with the `-h` argument). If you do that, then it prints a usage statement like that shown in Figure 7.1.

As the usage statement indicates, `config4cpp` provides the following commands: `parse`, `validate`, `dump`, `print`, `type`, `slist`, `llist` and `dumpSec`. Regardless of the command chosen, you must always use the `-cfg <source>` command-line argument to specify a *source* of configuration information. The source can be a file (specified with `file.cfg` or `file#file.cfg`) or a command (specified with `exec#...`) that, when executed, prints a configuration file to standard output. If the command contains spaces, then you need to enclose the command in double quotes, for example:

```
config4cpp -cfg exec#"curl -sS http://localhost/file.cfg" ...
```

### 7.1.2 Commonly Used Options

As discussed in Section 3.4 on page 22, many Config4* operations take two parameters that, when combined, specify the fully-scoped name of an item in a configuration file. For example:

```
logDir = cfg.lookupString("foo", "log.dir");
```

The first parameters (`"foo"`) specifies a *scope* and the second parameter (`"log.dir"`) specifies a local *name* within that scope. When using the `config4cpp` utility, you use the `-scope <...>` and `-name <...>` command-line options to specify the *scope* and *name* parameters for the underlying operations. For example:

```
config4cpp -cfg example.cfg print -scope foo -name log.dir
```

The `-scope <...>` and `-name <...>` options both default to empty strings. And since, internally, Config4* merges the two parameters to form a fully-scoped name, you can specify the fully-scoped name with either one of the two command-line options, and let the other option have its default value of an empty string. For example:

```
config4cpp -cfg example.cfg print -name foo.log.dir
```

As discussed in Section 3.8 on page 27, Config4* defines several constants that denote different types of entries found in a configuration file.

- `CFG_STRING`. A string variable.
- `CFG_LIST`. A list variable.

Figure 7.1: Usage statement for `config4cpp`

```
usage: config4cpp -cfg <source> <command> <options>

<command> can be one of the following:
  parse                Parse and report errors, if any
  validate             Validate <scope>.<name>
  dump                 Dump <scope>.<name>
  dumpSec              Dump the security policy
  print                Print value of the <scope>.<name> variable
  type                 Print type of the <scope>.<name> entry
  slist                List scoped names in <scope>.<name>
  llist                List local names in <scope>.<name>

<options> can be:
  -h                   Print this usage statement
  -set <name> <value> Preset name=value in configuration object
  -scope <scope>       Specify <scope> argument for commands
  -name <name>         Specify <name> argument for commands

  -secCfg <source>     Override default security policy
  -secScope <scope>    Scope for security policy

  -schemaCfg <source> Source that contains a schema
  -schema <full.name> Name of schema in '-schemaCfg <source>'

  -recursive           For llist, slist and validate (default)
  -norecursive         For llist, slist and validate
  -filter <pattern>    A filter pattern for slist and llist
  -types <types>       For llist, slist and validate

  -expandUid           For dump (default)
  -unexpandUid         For dump

<types> can be one of the following:
  string, list, scope, variables, scope_and_vars (default)

<source> can be one of the following:
  file.cfg       A configuration file
  file#file.cfg  A configuration file
  exec#<command> Output from executing the specified command
```

- `CFG_VARIABLES`. A variable, regardless of whether it is a string or a list.
- `CFG_SCOPE`. A scope.
- `CFG_SCOPES_AND_VARS`. A scope or a variable.

Some operations take one of the above values as a parameter to specify which type(s) of configuration entries the operation should process. When using `config4cpp`, you use the `-type <...>` command-line option to specify one of the above constants; however, you remove the `"CFG_"` prefix from the name of the constant and put the remaining part of the name in lower case. For example, `-type string` denotes the `CFG_STRING` constant. The default value of this option is `-type scopes_and_vars`.

The remaining sections of this chapter discuss each of the commands provided by `config4cpp`.

## 7.2 The `parse` Command

The `parse` command instructs `config4cpp` to parse the configuration file specified by `-cfg <source>` and then terminate. For example:

```
config4cpp -cfg example.cfg parse
```

If there is an error in the file, then `config4cpp` prints an error message before it terminates. In this way, the `parse` command provides a way to check for syntax errors in a recently created or modified configuration file.

If the file to be parsed is obtained by `-cfg exec#"..."`, then the default security policy (shown in Figure 5.1 on page 53) may not be permissive enough to allow the command to be executed. In such a case, you have two options.

If you just want to check whether the configuration file is syntactically valid, and you do not care about security policies, then you could execute the command yourself and save its output into a temporary file. Then you could run `config4cpp` on that temporary file:

```
command-that-prints-a-configuration-file > tmp.cfg
config4cpp -cfg tmp.cfg parse
```

Alternatively, if your aim is to check the suitability of a security policy, then you should create a file, say, `securityPolicy.cfg` that defines the three variables used to specify a security policy: `allow_patterns`, `deny_patterns` and `trusted_directories`. Then run `config4cpp` with the `-secCfg <source>` and `-secScope <scope>` command-line options:

```
config4cpp -cfg exec#"..." parse -secCfg securityPolicy.cfg \
           -secScope <scope>
```

The `-secScope <scope>` option specifies the scope that contains the three security-policy variables. If those variables are defined in the global scope then you can omit this command-line option because its value defaults to an empty string.

## 7.3 The `validate` Command

The `validate` command instructs `config4cpp` to parse a configuration file and then perform schema validation on a scope within the configuration file. If a validation error is encountered, then a descriptive error message is printed. This command may seem complex because its use requires a lot of command-line options. Because of this, I introduce it with an example.

Let's assume the file `myApplications.cfg` (Figure 7.2) contains configuration information for several applications, and you wish to perform schema validation for information in scope `foo` of that file. To do this, you will need to define a schema, such as that provided by the `example.fooSchema` entry in the `schemas.cfg` file (Figure 7.3).

You can perform the schema validation with the following command:

```
config4cpp -cfg myApplications.cfg validate \
           -scope foo \
           -schemaCfg schemas.cfg \
           -schema example.fooSchema \
           -recursive \
           -types scope_and_vars
```

Let's examine each command-line option. The schema validation is performed on the scope specified by the `-scope` option in the file specified by the `-cfg` option. The schema used is the list of strings provided by the variable specified by the `-schema` option in the file specified by the `-schemaCfg` file. The `-recursive` option specifies that the schema validator should recurse into nested scopes (such as `log`). The `types` option indicates whether the schema validator should perform validation checks for string variables (`-type string`), list variables (`-type list`), all variables (`-type variables`), scopes (`-type scope`), or everything (`-type scope_and_vars`).

The `-recursive` and `-types scope_and_vars` options are actually default values so they could have been omitted from the above example. If

Figure 7.2: The file myApplications.cfg

```
foo {
  timeout = "5 seconds";
  log {
    level = "2";
    dir = "/tmp";
  }
  colour = "green";
  price_list = [
    # item       colour    price
    #---------------------------------
     "apple",   "red",    "EUR 0.50",
     "widget",  "green",  "EUR 0.76",
     "pen",     "blue",   "USD 2.99"
  ];
  int_list = ["1", "2", "3"];
  temperature = "29 C";
}

bar {
  ... # details omitted for brevity
}
```

Figure 7.3: The file schemas.cfg

```
example.fooSchema = [
  "@typedef colour = enum[red, green, blue]",
  "@typedef temperature = float_with_units[C, F, K]",
  "@typedef money = units_with_float[USD, EUR, GBP]",
  "timeout     = durationSeconds",
  "log         = scope",
  "log.level   = int[0, 3]",
  "log.dir     = string[4, 4]",
  "colour      = colour",
  "price_list  = table[item,string, colour,colour, price,money]",
  "int_list    = list[int]",
  "temperature = temperature"
];

example.barSchema = [ ... ]; # details omitted for brevity
```

you use the -norecursive option, then the schema validation will examine only the specified scope—it will *not* recurse into nested scopes.

Figure 7.4 shows the algorithm used in config4cpp and config4j to implement the validate command. The code is straightforward. It initializes some variables from command-line options. Then it creates two (initially empty) Configuration objects and parses the files specified by the -cfg and -schemaCfg options. Finally, it uses a SchemaValidator object to parse the specified schema and perform schema validation.

Figure 7.4: Algorithm used by the validate command

```
cfgSource = ...   // from -cfg <...>
scope = ...       // from -scope <...> (default is "")
name = ...        // from -name <...> (default is "")
schemaSource= ... // from -schemaCfg <...>
schemaName = ...  // from -schema <...>
isRecursive = ... // from -recursive (default) or -norecursive
types = ...       // from -types <...> (default is scope_and_vars)
try {
  cfg = Configuration::create();
  sv = new SchemaValidator();
  schemaCfg = Configuration::create();
  cfg.parse(cfgSource);
  schemaCfg.parse(schemaSource);
  sv.parseSchema(schemaCfg.lookupList(schemaName, ""));
  sv.validate(cfg, scope, name, isRecursive, types);
} catch(ConfigurationException ex) {
  System.err.println(ex.getMessage());
  System.exit(1);
}
```

The purpose of showing you the code in Figure 7.4 is to illustrate that config4cpp and config4j are just thin wrappers around the corresponding Config4* libraries. This knowledge is important for application developers, because it means they can use config4cpp or config4j to "play with" Config4* and its API *without* needing to write any code. Doing this can shorten the learning curve.

## 7.4 The dump Command

When Config4* parses a configuration file, it stores information about scopes and *name=value* pairs in hash tables. Config4* provides a `dump()` operation that converts information in the hash tables into the syntax of a Config4* file. The `dump` command of the `config4cpp` utility is a thin wrapper around the `dump()` operation.

Figure 7.5 shows a configuration file called `foo.cfg`, and Figure 7.6 shows the output obtained when I ran the following command on a Linux machine:

```
config4cpp -cfg foo.cfg dump
```

Figure 7.5: The file `foo.cfg`

```
foo {
  # This is a comment
  timeout = "10 minutes";
  log {
    level = "1";
    @if (osType() == "windows") {
      dir = ".";
    } @else {
      dir = getenv("HOME") + "/.foo/logs";
    }
  }
}
```

Figure 7.6: Output of the `dump` command

```
foo {
  timeout = "10 minutes";
  log {
    dir = "/home/cjmchale/.foo/logs";
    level = "1";
  }
}
```

If you compare Figures 7.5 and 7.6, you may notice that the output of `dump` is different to the input file in several ways. First, comments are not preserved. Second, constructs that provide *adaptive configuration* (Section 2.11 on page 14)—such as `@if`-then-`@else` statements, function

calls and the concatenation operator ("+")—are not preserved. Finally, the order of items is not necessarily preserved, for example, the order of `foo.log.level` and `foo.log.dir` is swapped. These differences are a result of how Config4* works. When parsing a file, Config4* discards comments and fully evaluates all expressions so that when a *name=value* entry is stored in a hash table, the *value* is the *result* of evaluating an expression, rather than the expression itself. Finally, a hash table does not preserve the order in which items were added to it. Because of this, the `dump()` operation retrieves the items from a hash table in an unpredictable order, and it sorts the entries based on their names before processing them. This is why `dump()` outputs `foo.log.dir` before `foo.log.level`.

By default, the `dump` command dumps the contents of the root scope of a `Configuration` object. However, you can use the `-scope` and/or `-name` command-line options to instruct it to dump just a named scope or variable. For example,

```
config4cpp -cfg foo.cfg dump -scope foo.log
```

outputs the following:

```
foo.log {
  dir = "/home/cjmchale/.foo/logs";
  level = "1";
}
```

and:

```
config4cpp -cfg foo.cfg dump -name foo.log.dir
```

outputs the following:

```
foo.log.dir = "/home/cjmchale/.foo/logs";
```

The `-expandUid` and `-unexpandUid` options instruct `dump` how to process `uid-` entries. As an example of this, consider the `employees.cfg` file shown in Figure 7.7. The following command results in the output shown in Figure 7.8:

```
config4cpp -cfg employees.cfg dump -expandUid
```

The `-expandUid` option is actually the default, so it does not need to be explicitly stated. If you use the `-unexpandUid` option instead, then `uid-` entries are printed with unexpanded names.

Figure 7.7: The file `employees.cfg`

```
uid-employee {
  name = "John Smith";
  address = "...";
}
uid-employee {
  name = "Mary Jones";
  address = "...";
}
```

Figure 7.8: Result of dumping `employees.cfg`

```
uid-000000000-employee {
  address = "...";
  name = "John Smith";
}
uid-000000001-employee {
  address = "...";
  name = "Mary Jones";
}
```

## 7.5 The `dumpSec` Command

The name of the `dumpSec` command is an abbreviation for "dump security". This command displays the `allow_patterns`, `deny_patterns` and `trusted_directories` of the security policy. Usually, the items displayed will be those of the *default* security policy (see Figure 5.1 on page 53). However, recall from Section 7.2 on page 68, that you can use the `-secCfg` and `-secScope` options of `config4cpp` to specify a different security policy.

## 7.6 The `print` Command

The `print` command prints the value of a configuration variable specified by the `-scope` and/or `-name` options. For example, recall the `foo.cfg` file shown earlier (Figure 7.5 on page 72). The following command:

```
config4cpp -cfg foo.cfg print -name foo.log.dir
```

displays:

```
/home/cjmchale/.foo/logs
```

The `print` command differs from the `dump` command in two ways. First, you can `print` a variable, but you can `dump` a variable *or* a scope. Second, `print` displays only the *value* of a variable, but `dump` displays *name=value* in Config4* syntax. For example, the following command:

```
config4cpp -cfg foo.cfg dump -name foo.log.dir
```

displays:

```
foo.log.dir = "/home/cjmchale/.foo/logs";
```

The `print` command enables a UNIX shell script to access configuration variables in a Config4* file. For example, the following line in a shell script will create the directory specified by the `foo.log.dir` variable in the `foo.cfg` file: script:

```
mkdir -p ‘config4cpp -cfg foo.cfg print -name foo.log.dir‘
```

If you `print` a list variable, then each each item in the list is printed on a separate line. For example, assume the `bar.cfg` file contains the following line:

```
file_list = ["tmp.txt", "TO_DO.txt", "make.log"];
```

The following command:

```
config4cpp -cfg bar.cfg print -name file_list
```

displays:

```
tmp.txt
TO_DO.txt
make.log
```

This one-item-displayed-per-line property makes it easy for a UNIX shell script to process each item in a list, for example:

```
for file in ‘config4cpp -cfg bar.cfg print -name file_list‘
do
  echo Processing $file
done
```

## 7.7 The `type` Command

The `type` command displays the type (`string`, `list` or `scope`) of an entry in a configuration file. For example:

```
config4cpp -cfg foo.cfg type -name foo
```

displays:

```
scope
```

and

```
config4cpp -cfg foo.cfg type -name foo.log.level
```

displays:

```
string
```

If the specified item does not exist in the configuration file then the `type` command displays:

```
no_value
```

## 7.8 The `slist` and `llist` Commands

The `slist` command is a wrapper around `listFullyScopedNames()`. Likewise, `llist` is a wrapper around `listLocallyScopedNames()`.

These commands list the fully- or locally-scoped names of entries in the scope specified by the `-scope <...>` and `-name <...>` command-line options. If you let those options have their default values (an empty string), then the commands will display a sorted list of entries in the root scope of the configuration file.

The `-types <...>` option specifies the types of items that the commands should list. For example, `-types variables` lists the names of variables, while `-types scope` lists the names of scopes. The default value of this option is `scope_and_vars`, which lists both variables and scopes.

The `-recursive` and `-norecursive` options specify whether the commands should recurse into nested scopes to list their entries. The default value is `-recursive`.

The following examples are based on the `example.cfg` configuration file in Figure 7.9.

The following command:

```
config4cpp -cfg example.cfg slist -scope example.foo
```

lists the entries in the `example.foo` scope and (recursively) in nested scopes:

Figure 7.9: The file `example.cfg`

```
example {
  foo {
    timeout = "infinite";
    log {
      dir = "/tmp";
      level = "1";
    }
  }
  bar {
    greeting = "Hello, world";
  }
}
```

```
example.foo.log
example.foo.log.dir
example.foo.log.level
example.foo.timeout
```

If you change `slist` to `llist` in the above command, and re-run it, then the output will be as follows.

```
log
log.dir
log.level
timeout
```

One of the parameters passed to the `listFullyScopedNames()` and `listLocallyScopedNames()` operations is an array of strings that specify *filter patterns*. Each filter pattern is a string in which `"*"` is a wildcard that matches zero or more characters. An entry will be included in the returned list only if: (1) the filters patterns array is empty (thus indicating that no filtering is performed); or (2) the (unexpanded form of the) entry's name matches at least one of the filter patterns.

You can use the `-filter <...>` command-line option to specify a filter. You can use this option multiple times to specify multiple filters.

Pattern filters can be very useful if you are writing an application that makes use of *uid* entries in a configuration file. As an example, consider a `people.cfg` configuration file that contains a mixture of `uid-friend` and `uid-enemy` entries within the `people` scope. The following command will list just the `uid-friend` entries:

```
config4cpp -cfg people.cfg llist -scope people -filter uid-friend
```

The following command will list all the "uid-" entries:

```
config4cpp -cfg people.cfg llist -scope people -filter "uid-*"
```

## 7.9 Summary

The config4cpp and config4j utilities provide command-line wrappers for operations in the Config4* libraries. These utilities serve a few purposes.

First, when you have written or edited a configuration file, you can use the parse command to check if the file has any syntax errors or, use the validate command to check it for schema validation errors.

Second, the utilities provide a way for you to experiment with the Config4* API without having to write code. As such, these utilities can shorten the learning curve for developers. For example:

- The -secCfg and -secScope options enable you to experiment with defining your own security policies.
- The validate command enables you to explore the syntax and semantics of the schema language.
- The dump command enables you to check if adaptive configuration constructs behave the way you think they should.
- The slist and llist commands, and their -filter option, provide a way for you to experiment with the listFullyScopedNames() and listLocallyScopedNames() operations. This can be useful if you need to implement a browser-type application or if you plan to work with *uid* entries.

Finally, the print command make it possible for a UNIX shell script to retrieve information from a Config4* file. This makes it possible to use Config4* to configure shell script-based applications.

# Part III

# Full Details of Syntax

# Chapter 8

# Configuration File Syntax

## 8.1 Introduction

This chapter discusses all the syntax acceptable in configuration files. Figure 8.1 provides a formal grammar for *most* of the syntax but, for brevity, the grammar omits some definitions. For example, the lexical definition of comments, strings and identifiers are discussed in text rather than being defined in the grammar of Figure 8.1. Likewise, the string and list functions (denoted by `StringFunction` and `ListFunction` in the grammar) are discussed in text rather than being defined in the grammar.

## 8.2 Comments

A comment starts with the "`#`" character and continues until the end of the line, as shown in the example below:

```
# This is a comment
```

Comments are removed by the lexical analyser, which is why they are not mentioned in the formal grammar in Figure 8.1.

## 8.3 Strings

There are two ways to write a `STRING`.

Figure 8.1: Formal grammar of Config4* syntax

Notation: | denotes choice, [...] denotes an optional component, {...}* denotes 0 or more repetitions, and (...) denotes grouping.

```
configFile     = StmtList
StmtList       = { Stmt }*
Stmt           = IDENTIFIER ( "=" | "?=" | "+=" ) StringExpr ";"
               | IDENTIFIER ( "=" | "?=" | "+=" ) ListExpr ";"
               | IDENTIFIER "{" StmtList "}" [ ";" ]
               | "@include" StringExpr [ "@ifExists" ] ";"
               | "@copyFrom" IDENTIFIER [ "@ifExists" ] ";"
               | "@remove" IDENTIFIER ";"
               | "@error" StringExpr ";"
               | "@if" "(" Condition ")" "{" StmtList "}"
                 { "@elseIf" "(" Condition ")" "{" StmtList "}" }*
                 [ "@else" "{" StmtList "}" ]
                 [ ";" ]
StringExpr     = String { "+" String }*
String         = STRING
               | IDENTIFIER
               | StringFunction
ListExpr       = List { "+" List }*
List           = "[" StringExprList [ "," ] "]"
               | IDENTIFIER
               | ListFunction
StringExprList = empty
               | StringExpr  "," StringExpr *
Condition      = OrCondition
OrCondition    = AndCondition { "||" AndCondition }*
AndCondition   = TermCondition { "&&" TermCondition }*
TermCondition  = [ "!" ] "(" Condition ")"
               | StringExpr "==" StringExpr
               | StringExpr "!=" StringExpr
               | StringExpr "@in" ListExpr
               | StringExpr "@matches" StringExpr
```

The first way is as a sequence of characters enclosed within double quotes. Within such a string, "%" acts as an escape character. The recognized escape sequences are as follows. %n denotes a newline character. %t denotes a TAB character. %" denotes a double quote. %% denotes a percent sign. Many programming languages use a backslash ("\") as an escape character so the use of "%" may seem strange

to some people. However, in my experience, using "\" as an escape character results in awkwardness when writing Windows-style directory names, such as C:\temp\foo.txt, which normally have to be written as C:\\temp\\foo.txt. Config4* uses "%" as the escape character to avoid this problem.

The second way to write a string is as a (possibly multi-line) sequence of characters enclosed between <% and %>. No escape sequences are recognised between <% and %>. If the <%...%> notation seems familiar to some readers it is because this notation is borrowed from Java Server Pages (JSP). The <%...%> notation is useful if you want to embed, say, a code segment in a configuration file.

You can combine both forms of string by using the string concatenation ("+") operator.

```
big_string = <%
    ... // some Java code
%> + "<%" + <%
    ... // some more Java code
%>
```

## 8.4 Identifiers

An IDENTIFIER is a sequence of one or more of the following characters: upper- or lower-case letters, digits, a minus sign ("-"), an underscore ("_"), a colon (":"), a period (".), a dollar sign ($), a question mark ("?"), a forward slash ("/") or a backslash ("\"). There are two comments to be made about this range of allowable characters.

First, one goal of Config4* is to support internationalization, so accented characters (such as "á" and "ö") and ideographs are permitted in an IDENTIFIER. Likewise, the digits permitted in an IDENTIFIER include the Roman digits ("0" through to "9") as well as digits used in other scripts.[1]

Second, a Config4* IDENTIFIER should be able to support names not just in many *human* languages, but also names in many *computer* languages. For example, ensuring that Foo$Bar, X::Y::Z, and done? are valid identifiers makes it possible for a Config4* file to store meta-data about applications written in many popular programming languages,

[1] Readers should be forewarned that some implementations of Config4* may have *incomplete* internationalization support. You can find a discussion of this in the *Config4* Maintenance Guide*.

such as C++, Java, Perl and Ruby. Likewise, permitting `"/"` and `"\"` in identifiers enables a Config4* file to contain meta-data about file names and (a useful subset of) URLs.

Config4* applies special treatment to any identifier that starts with `"uid-"`, for example, `uid-foo`. The `"uid-"` prefix denotes a *unique identifier*; you can read the motivation for such identifiers in Section 2.12 on page 15. Config4* modifies the name of a `"uid-"` prefixed variable by inserting a sequence of nine digits and `"-"` after `"uid-"`. For example, `uid-foo` might be changed to `uid-000000042-foo`. The nine-digit number starts at zero and is incremented by one for every encounter of an identifier that has a `"uid-"` prefix.

If Config4* encounters an identifier starting with `"uid-<digits>-"`, then the digits are *replaced* with a newly generated nine-digit number. This is to ensure correct behaviour in pathological cases such as the following. Consider a configuration file that contains multiple `uid-foo` identifiers. If this file is parsed and the `dump()` operation is used to save the parsed file to, say, `expanded-uid.cfg`, then the newly written file may contain identifiers of the form `uid-<digits>-foo`. Now consider another file of the form:

```
uid-foo { ... };
uid-foo { ... };
uid-foo { ... };
@include "expanded-uid.cfg";
```

When parsing the above file, it is necessary to replace the digits of the `uid-foo` entries contained in the `expanded-uid.cfg` file to ensure they do not conflict with the expanded form of the `uid-foo` entries defined before the `@include` command.

## 8.5 Assignment Statements

An *unconditional assignment* statement takes one the form:

```
name = value;
```

A *conditional assignment* statement takes the form:

```
name ?= value;
```

A conditional assignment statement assigns a value to the specified variable *only if* the variable does not already have a value.

An *append assignment* statement takes the form:

```
name += value;
```

An append assignment statement appends the specified value to an already-existing variable.

Note that all three forms of the assignment statement are terminated with a semicolon (";"). A `value` can be either a *string* or a *list* of comma-separated strings inside matching "[" and "]":

```
local_domain = "bar.com";          # a string
some_fonts = ["Times", "Courier"]; # a list
some_fonts += ["Garamond"];
```

You can use the "+" operator to concatenate strings and lists.

```
host = "foo." + local_domain;
all_fonts = some_fonts + ["Ariel", "Symbol"];
```

The above example also illustrates that one variable can be defined in terms of a previously defined variable. For example, the `host` variable is defined by concatenating together a string and the `local_domain` variable.

## 8.6 Scopes

A configuration file can contain named scopes. The following example defines a scope called `server` that contains several assignment statements.

```
server {
    name              = "bankSrv";
    timeout           = "2 minutes";
    diagnostics_level = "2";
}
```

You can optionally place a semicolon after the closing "}" of a scope. The reason for this is that a scope looks a bit like a class definition in C++ or Java. A semicolon appears after the class definition in C++ but not in Java.

```
class Foo { ... }; // C++
class Bar { ... }  // Java
```

Being flexible about whether or not a semicolon follows the closing "}" of a scope makes it easy for people who come from a C++ *or* Java background.

You *cannot* use an `@include` statement (discussed in Section 8.7) inside a scope. Instead, `@include` statements can be used only in the global scope.

The *fully scoped* name of a variable is its *local* name prefixed by the name of its enclosing scope and separated by `"."`. In the example at the start of this section, the fully scoped name of `timeout` is `server.timeout`. Use of scopes enables users to type *local* (that is, the short form of) names rather than the longer, *fully scoped* names. At the start of this section was an example that made use of a scope. That example is equivalent to the following, more verbose example, which does not use scopes:

```
server.name              = "bankSrv";
server.timeout           = "2 minutes";
server.diagnostics_level = "2";
```

You can re-open scopes and nest them arbitrarily. For example:

```
outer {
    inner {
        foo = "Hello, world";
    };
};
outer.inner { # re-opening of scope
        bar = "Goodbye, world";
};
```

When a variable is used in an expression, the search for that variable usually starts at the current scope and works outwards. You can override this search order by prefixing the variable with a dot; this instructs Config4* to look for the specified variable in the global scope. For example, the value of `outer.inner.food_1` below is `"apples and oranges"`, while the value of `outer.inner.food_2` is `"apples and bananas"`.

```
fruit = "bananas";
outer {
    fruit = "oranges";
    inner {
        food_1 = "apples and " + fruit;
        food_2 = "apples and " + .fruit;
    };
};
```

## 8.7 The `@include` Statement

An `@include` statement instructs Config4* to parse the specified configuration file.

```
@include "/tmp/foo.cfg";
```

By default, `@include` reports an error if the specified file does not exist. However, if you place `"@ifExists"` at the end of an `@include` statement then `@include` does not complain about a non-existent file.

```
@include "/tmp/foo.cfg" @ifExists;
```

The `@include` command can parse not just files, but also the output of executing an external command. This is done by using a string of the form `"exec#command"` as an argument to `@include`.

```
@include "exec#curl -sS http://localhost/someFile.cfg";
```

By default, `@include` reports an error if the specified command exits with an error status. You can instruct Config4* to ignore the unsuccessful execution of an `@include` command by placing `"@ifExists"` at the end of the `@include` statement.

```
@include "exec#curl -sS http://localhost/someFile.cfg" @ifExists;
```

Version 1.2 of Config4J introduces an additional, and Java-specific, form of the `@include` statement, in which the file to be included is specified on the classpath.

```
@include "classpath#path/to/file.cfg";
```

## 8.8 The `@copyFrom` Statement

The `@copyFrom` statement takes the following form:

```
@copyFrom "scope";
```

This command copies all the variables and nested scopes from the specified `scope` into the current scope. The typical use of this command is to copy default values from one scope into several other scopes, as Figure 8.2 shows.

In this example, the `acme.defaults` scope contains all the configuration variables likely to have similar values in most of the applications (denoted by the scopes `acme.app_1`, `acme.app_2` and `acme.app_3`). Then the scope for a particular application, for example, `acme.app_1`, uses the `@copyFrom` command to copy the values from the `acme.defaults` scope. Notice that the `acme.app_2` and `acme.app_3` scopes copy all the values from the `acme.defaults` scope and then selectively override some values.

Figure 8.2: Examples of the `@copyFrom` statement

```
acme {
    defaults {
        log {
            dir   = "C:\acme\logs";
            level = "0";
        };
        timeout = "2 minutes";
        thread_pool_size = "5";
    };
    app_1 {
        @copyFrom "acme.defaults";
    };
    app_2 {
        @copyFrom "acme.defaults";
        log.level = "1";
    };
    app_3 {
        @copyFrom "acme.defaults";
        thread_pool_size = "10";
    };
};
```

When using the `@copyFrom` statement, you must specify the *fully scoped* name of the scope to be copied. For example, the `@copyFrom` statements in Figure 8.2 specify the scope as `acme.defaults` rather than as just `defaults`. If a configuration file contains deeply nested scopes, then specifying the fully scoped name of a scope to be copied can result in undesirable verbosity. However, Section 8.12.6 on page 95 explains how the `siblingScope()` function can reduce such verbosity.

By default, `@copyFrom` reports an error if the specified scope does not exist. However, if you place `"@ifExists"` at the end of an `@copyFrom` statement then `@copyFrom` does not complain about a non-existent scope.

```
@copyFrom "acme.defaults" @ifExists;
```

The `@ifExists` form of the `@copyFrom` command can be used to override some variables based on, for example, the operating system, the user running the application or the host on which the application is running.

```
override.pizza { ... }
override.pasta { ... }
fooSrv {
```

```
   # Set default values
   ...
   # Modify some values for particular hosts
   @copyFrom "override." + exec("hostname") @ifExists;
}
```

## 8.9 The @if-then-@else Statement

Figure 8.3 shows some examples of @if-then-@else statements.

Figure 8.3: Configuration file with advanced features

```
 1  production_hosts = ["pizza", "pasta", "zucchini"];
 2  test_hosts       = ["foo", "bar", "widget", "acme"];
 3
 4  @if (exec("hostname") @in production_hosts) {
 5      server_x.port = "5000";
 6      server_y.port = "5001";
 7      server_z.port = "5002";
 8  } @elseIf (exec("hostname") @in test_hosts) {
 9      server_x.port = "6000";
10      server_y.port = "6001";
11      server_z.port = "6002";
12  } @else {
13      @error "This is not a production or test machine";
14  }
15  @if (osType() == "windows") {
16      tmp_dir = replace(getenv("TMP"), "\", "/");
17  } @else {
18      tmp_dir = "/tmp";
19  }
```

The conditions used in @if-then-@else statements can be in any of the following formats.

- "string" == "another string"
- "string" != "another string"
- "string" @in ["a", "list", "of", "string"]
- "string" @matches "pattern". Within the pattern, "*" is a wildcard that matches zero or more characters. For example, the condition "hello" @matches "*lo" evaluates to true.

- `condition && condition`. This is the boolean AND of two conditions.
- `condition || condition`. This is the boolean OR of two conditions.
- `(condition)`. The parenthesis are used for grouping.
- `!(condition)`. This is the negation of a condition.

## 8.10 The `@error` Statement

The `@error` statement instructs Config4* to stop parsing and instead report an error.

```
@error "Something has gone wrong";
```

Config4* reports the error by throwing an exception back to application code. The application code should communicate the exception's text message to the user, for example, by writing the text message to a console or displaying it in a GUI dialog box.

## 8.11 The `@remove` Statement

The `@remove` statement removes a previously-defined variable or scope. To see why the `@remove` command might be useful, let us assume you want to specify the full path names of several log files that happen to reside in the same directory. It would be tedious to write the full path name of the directory for each log file. Instead, you can can define a temporary variable called, say, `_log_dir` and used it as follows:

```
_log_dir = "/path/to/log/dir";
app1_log_file = _log_dir + "/app1.log";
app2_log_file = _log_dir + "/app2.log";
app3_log_file = _log_dir + "/app3.log";
@remove _log_dir;
```

A useful convention shown in the above example is to use an underscore (`"_"`) at the start of the name of a temporary variable. This makes it easy to see which variables are "normal" variables and which are temporary ones that will be removed later.

You may be wondering why temporary variables should be removed at all. There are two reasons for this. First, unneeded variables clutter

up a configuration file and so can cause confusion for users. Second, by insisting a configuration file contain *only required* variables, an application can make use of a schema validator (Chapter 9) that can perform extensive error checking on the contents of a configuration file.

## 8.12 Functions

Table 8.1 lists the functions that Config4* provides.

Table 8.1: Config4* functions

| Function | Return type | Section |
|---|---|---|
| `configFile()` | string | 8.12.5 |
| `configType("name")` | string | 8.12.6 |
| `exec("command")` | string | 8.12.3 |
| `exec("command", "default value")` | string | 8.12.3 |
| `fileToDir("/path/to/file.txt")` | string | 8.12.5 |
| `getenv("name")` | string | 8.12.2 |
| `getenv("name", "default value")` | string | 8.12.2 |
| `isFileReadable("fileName.txt")` | boolean | 8.12.6 |
| `join(["list", "of", "string"], " ")` | string | 8.12.4 |
| `osDirSeparator()` | string | 8.12.1 |
| `osPathSeparator()` | string | 8.12.1 |
| `osType()` | string | 8.12.1 |
| `readFile("/path/to/file.txt")` | string | 8.12.5 |
| `replace("\a\b\c", "\", "/")` | string | 8.12.4 |
| `siblingScope("name")` | string | 8.12.6 |
| `split("red green blue", " ")` | list | 8.12.4 |

Config4* considers the opening `"("` to be part of a function name, so you cannot place a space before it. For example, Config4* accepts the first statement below but reports an error for the second statement:

```
x = configFile();  # okay
y = configFile (); # error
```

Treating the opening `"("` as being part of a function's name might seem strange, but Config4* does this to guarantee that the names of functions do not conflict with the names of variables or scopes. This makes it possible for future versions of Config4* to provide additional

functions without any risk of the newly added functions causing problems for existing configuration files.

The following subsections discuss the functions in logical groupings.

### 8.12.1 Querying the Operating System

Some of the built-in functions have names starting with `"os"`, which indicates they return information about the operating system environment.

The `osType()` function returns `"windows"` if you are running on a Microsoft Windows-based computer, and `"unix"` if you are running on a UNIX-based computer.

The `osDirSeparator()` function returns the character that the operating system uses as a directory separator. This is `"\"` on Windows and `"/"` on UNIX.

The `osPathSeparator()` function returns the character that the operating system uses to separate a list of directories. This is `";"` on Windows and `":"` on UNIX.

### 8.12.2 Accessing Environment Variables

The `getenv()` function enables you to access an environment variable. This function can take either one or two parameters. The first parameter is the name of the environment variable to access:

```
example = getenv("FOO_HOME");
```

The second (and optional) parameter to this function is a default value that is used if the specified environment variable does not exist:

```
example = getenv("FOO_HOME", "/tmp");
```

If you do not specify a default value and the specified environment variable does not exist then Config4* reports an error:

```
someFile.cfg, line 12: cannot access the 'FOO_HOME'
environment variable
```

### 8.12.3 Executing External Commands

The `exec()` function executes an external command and returns whatever text that command writes to its standard output. This function can take either one or two parameters. The first parameter is the external command to execute, as the following examples illustrate:

```
example_1 = exec("hostname");
example_2 = exec("ls /tmp");
example_3 = exec("ls " + getenv("HOME", "/") );
```

The second (and optional) parameter to this function is a default value that is used if Config4* cannot successfully execute the specified external command:

```
example = exec("hostname", "localhost");
```

If you do not specify a default value and Config4* cannot successfully execute the specified external command then Config4* reports an error:

```
someFile.cfg, line 3: exec("ls /x/y/z") failed:
ls: /x/y/z: No such file or directory
```

### 8.12.4 Manipulating Strings and Lists

The example below illustrates the `split()` and `join()` functions:

```
colours_and_spaces = "red green blue";
colour_list = split(colours_and_spaces, " ");
colours_and_commas = join(colour_list, ",");
```

The `split()` function takes two parameters. The first parameter is a string to be broken up into a list of smaller strings. The second parameter indicates a search string; the first string is broken into list elements at each occurrence of this search string. In the above example, `colour_list` is assigned the value `["red", "green", "blue"]`.

The `join()` function is the opposite of `split()`. It takes two parameters; the first parameter is a list and the second parameter is a string. The `join()` function concatenates all the elements of the list using the string as a separator. In the above example, `colours_and_commas` is assigned the value `"red,green,blue"`.

In the above example, the overall effect of using `split()` and `join()` is to *replace* all spaces in a string with commas. To make this easier, Config4* provides a `replace()` function.

```
colours_and_commas=replace("red green blue", " ", ",");
```

The `replace()` function takes three string parameters: *original*, *search* and *replacement*. This function replaces all occurrences of the *search* string in the *original* string with the *replacement* string.

### 8.12.5 Files and Directories

The `configFile()` function does not take any parameters; it returns the name of the configuration file being parsed.

The `fileToDir()` function takes one parameter—the name of a file—and returns the name of the directory in which that file resides. The returned directory name is guaranteed to not have `"/"` or `"\"` at the end. For example, `fileToDir("/tmp/foo.cfg")` returns `"/tmp"`. As the table in Table 8.2 shows, the `fileToDir()` function works even for boundary cases, such as for files in the root directory of a file system.

Table 8.2: Example results of calling `fileToDir()`

| `filename` | `fileToDir(filename)` |
|---|---|
| `"/tmp/foo.cfg"` | `"/tmp"` (UNIX and Windows) |
| `"C:\tmp\foo.cfg"` | `"C:\tmp"` (Windows only) |
| `"foo.cfg"` | `"."` (UNIX and Windows) |
| `"/foo.cfg"` | `"/."` (UNIX and Windows) |
| `"\foo.cfg"` | `"\."` (Windows only) |
| `"C:\foo.cfg"` | `"C:\."` (Windows only) |

The combination `fileToDir(configFile())` returns the directory in which the configuration file being parsed resides. This can be useful if you want to write a top-level configuration file that includes other configuration files that reside within the same directory.

```
@include fileToDir(configFile()) + "/file1.cfg";
@include fileToDir(configFile()) + "/file2.cfg";
@include fileToDir(configFile()) + "/file3.cfg";
```

This technique can work even if the configuration file is hosted on a web server and is being accessed through the `curl` utility. To see why, let's assume the top-level configuration file is specified as:

```
exec#curl -sS http://myHost/foo/foo.cfg
```

Config4* will execute that command and then parse its output. During this parsing, the `configFile()` function returns:

```
exec#curl -sS http://myHost/foo/foo.cfg
```

The `fileToDir()` function does not check that its parameter is a valid file name; rather it just trims its parameter back to the last occurrence of `"/"`, so the result of `fileToDir(configFile())` is:

```
exec#curl -sS http://myHost/foo
```

The first `@include` statement in the example appends `"/file1.cfg"`, so the `@include` statement becomes:

```
@include "exec#curl -sS http://myHost/foo/file1.cfg";
```

One thing to keep in mind is that downloading a multi-part configuration file from a web server will be slower than downloading a monolithic configuration file. It will probably take just a fraction of a second longer to download the multi-part configuration file, so you might think that such an overhead is insignificant. However, in a large organization there might be thousands of users downloading their applications' configuration files from the same web server. In such an organization, all those fractions of a second extra overhead might add up to be a significant overhead.

### 8.12.6 Miscellaneous Functions

The `configType()` function takes a string parameter that specifies the fully-scoped name of an entry in the configuration file. It returns the value `"string"` if the entry is a string variable, `"list"` if the entry is a list variable, `"scope"` if the entry is a scope, or `"no_value"` if there is no such entry.

The `isFileReadable()` function takes a string parameter that specifies the name of a file. It returns `true` if the file exists and is readable; it returns `false` otherwise. An example of the intended use of this function is shown below:

```
files_to_process = ["file1.txt", "file2.txt", "file3.txt"];
@if (isFileReadable("file4.txt")) {
    files_to_process = files_to_process + ["file4.txt"];
}
```

The `siblingScope()` function takes a string parameter that specified the local name of a scope that is a sibling of the current scope. It returns the fully scoped name of the specified scope. This function is provided to simplify a common use case of the `@copyFrom` statement that is shown in Figure 8.4.

It is common for the `@copyFrom` statement to be used to copy the contents of a scope that is at the same level of nesting—what I call a *sibling* scope. If the sibling scope is deeply nested in the configuration file, then, as shown in Figure 8.4, the `@copyFrom` statement can be quite

Figure 8.4: Verbose `@copyFrom` statements

```
acme.uk.london.sales {
    defaults {
        timeout = "2 minutes";
        log.level = "1";
    }
    app1 {
        @copyFrom "acme.uk.london.sales.defaults";
    }
    app2 {
        @copyFrom "acme.uk.london.sales.defaults";
       log.level = "0";
    }
    app3 {
        @copyFrom "acme.uk.london.sales.defaults";
       log.level = "0";
    }
}
```

verbose. If, later on, the scope hierarchy is renamed (perhaps by being copy-and-pasted to another part of the configuration file), then all the `@copyFrom` statements will have to be updated to specify the renamed sibling scope. Doing this has be tedious and error-prone.

Figure 8.5 shows the configuration file after it has been modified to make use of the `siblingScope()` function. The `@copyFrom` statements in this modified file are more concise and easier to visually verify for correctness. In addition, if the `acme.uk.london.sales` scope is renamed, then the `@copyFrom` statements will continue to work without any need for updating.

Figure 8.5: Using `siblingScope()` to get concise `@copyFrom` statements

```
acme.uk.london.sales {
    defaults {
        timeout = "2 minutes";
        log.level = "1";
    }
    app1 {
        @copyFrom siblingScope("defaults");
    }
    app2 {
        @copyFrom siblingScope("defaults");
       log.level = "0";
    }
    app3 {
        @copyFrom siblingScope("defaults");
       log.level = "0";
    }
}
```

# Chapter 9

# The Config4* Schema Language

## 9.1 Introduction

A *schema* is a blueprint or definition of a system. For example, a database schema defines the layout of a database: its tables, the columns within those tables, and so on. It is common (but not a requirement) for a schema to be written in the same syntax as the system it defines. For example, a database's schema might be stored within a table of the database itself.

Another technology that uses schemas is XML. The first schema language for XML was called *document type definition* (DTD). Many people felt DTD was sufficient to define schemas for text-oriented XML documents, which tend to have a simple structure, but not flexible enough to define schemas for more structured, data-oriented XML documents. Because of this, several competing XML schema languages were defined, including XML Schema and RELAX NG.

By itself, a schema it not very useful; you also need to have a piece of software, called a *schema validator*, that can compare a system (database, XML file or whatever) against the system's schema definition and report errors. Within the Config4* library is a class called `SchemaValidator` that, as its name suggests, implements a schema validator. Application developers can use this to automate useful validity checks on configuration information.

As you read this chapter, you may wish to test some of the examples to ensure you fully understand the semantics of the schema language. You do not need to write any code to do such testing. Instead, you can use the `validate` command of the `config4cpp` or `config4j` utilities to test a schema. Section 7.3 on page 69 discusses how to use that command.

## 9.2 Syntax

A Config4* schema consists of an array of strings. The grammar for a string within a schema is shown in Figure 9.1. As can be seen, a string within a schema can be one of the following: an *identifier rule*, an *ignore rule* or a *type definition*.

Figure 9.1: Formal grammar of the Config4* schema language

Notation: `|` denotes choice, and `{...}*` denotes 0 or more repetitions.

```
stringInSchema    = idRule | ignoreRule | TypeDefinition

idRule            = OptOrReq Name '=' TypeName OptArgList

OptOrReq          = empty | '@optional' | '@required'

ignoreRule        = '@ignoreScopesIn'     Name
                  | '@ignoreVariablesIn'  Name
                  | '@ignoreEverythingIn' Name

TypeDefinition    = '@typedef' TypeName '=' TypeName OptArgList

OptArgList        = empty | '[' ArgList ']'
ArgList           = empty | Arg { ',' Arg }*
Arg               = IDENTIFIER | STRING

Name              = IDENTIFIER
TypeName          = IDENTIFIER
```

### 9.2.1 Identifier Rules

An *identifier rule* specifies the permissible type of an item of configuration information. This can be illustrated with an example. Let's assume

you are writing an application that requires configuration information like that shown in Figure 9.2. A suitable schema for this configuration is shown (in Java syntax) in Figure 9.3. Each string in that schema is an identifier rule; it specifies the permitted type for a named item of configuration information. For example, the first rule in Figure 9.3 specifies that `timeout` is of type `durationMilliseconds`, and the fourth rule specifies that `log` is a scope.

Figure 9.2: Example configuration for an application

```
timeout = "2 minutes";
fonts = ["Times Roman", "Helvetica", "Courier"];
background_colour = "white";
log {
    dir = "C:\foo\logs";
    level = "1";
}
```

Figure 9.3: Schema for the example configuration shown in Figure 9.2

```
String[] schema = new String[] {
    "timeout = durationMilliseconds",
    "fonts = list[string]",
    "background_colour = enum[grey, white, yellow]",
    "log = scope",
    "log.dir = string",
    "log.level = int[0, 3]"
};
```

The simplest form of an identifier rule is *name=type*. The *type* can be optionally followed by a list of arguments. The use of arguments is illustrated by the rules for `fonts`, `log.level` and `background_colour` in Figure 9.3. In the rule for `fonts`, the argument specifies that each item in the `fonts` list should be interpreted as a `string`, rather than, say, a `boolean` or `int`. In the rule for `log.level`, the arguments specify minimum and maximum values for the integer. In the rule for `background_colour`, the `enum` type specifies an enumeration of allowable values, which is indicated by its list of arguments.

### 9.2.2 The `@optional` and `@required` Keywords

You can optionally use one of the keywords `@optional` or `@required` at the start of an identifier rule. For example:

```
String[] schema = new String[] {
    "x = string", // defaults to @optional
    "@optional y = string",
    "@required z = string"
};
```

If you do not specify one of those keywords, then the default behaviour is as if you had specified `@optional`.

The semantics of `@required` are that the specified entry *must* be present in the configuration scope being validated. Conversely, the semantics of `@optional` are that the specified entry *may* be (but is *not* required to be) in the scope being validated.

The default semantics of entries being optional means that a schema works well with both fallback configuration (Section 3.6.3 on page 25) and default parameters passed to lookup-style operations (Section 3.7 on page 27).

The semantics of `"uid-"` entries are that they may appear zero or more times. Because of this, `"uid-"` entries are intrinsically optional. If you try to use `@required` with a `"uid-"` entry, then the schema validator throws an exception message.

### 9.2.3 Defining a New Type

A *type definition* defines a new type in terms of an existing type. As an example of this, Figure 9.4 shows a revised schema for the configuration previously shown in Figure 9.2. This revised schema defines two new types, `colour` and `logLevel`, and then uses them to specify the types of the `background_colour` and `log.level` variables.

The ability to define new types serves two purposes. First, it helps to ensure consistency if you need to use a type—such as `colour` or `logLevel`—for several variables in a schema. Second, and more importantly, it enables you to work around a limitation in the syntax of the schema language. To understand this, let's assume the `colour_list` variable in a configuration file specifies a list of colours. You *cannot* specify this in a schema with the following:

```
String[] schema = new String[] {
    "colour_list = list[enum[gray, white, yellow]]"
};
```

Figure 9.4: Alternative schema for the example configuration shown in Figure 9.2 on page 101

```
String[] schema = new String[] {
    "@typedef colour = enum[gray, white, yellow]"
    "@typedef logLevel = int[0, 5]",
    "timeout = durationMilliseconds",
    "fonts = list[string]",
    "background_colour = colour",
    "log = scope",
    "log.dir = string",
    "log.level = logLevel"
};
```

This is because the schema syntax does not permit the nesting of argument lists. You can work around this syntactic limitation with the aid of a `@typedef` statement, as shown below:

```
String[] schema = new String[] {
    "@typedef colour = enum[gray, white, yellow]",
    "colour_list = list[colour]"
};
```

### 9.2.4 Available Schema Types

A complete list of the built-in schema types are shown in Table 9.1.

One of the types in Table 9.1 is `scope`, which, as its name suggests, indicates that an entry in a configuration file is a scope. All the remaining types in the table fall into two categories: string-based types and list-based types. I discuss those in the following subsections.

#### 9.2.4.1 String-based Types

The `boolean` type is a string in which only the values `"true"` and `"false"` are valid. The `boolean` type does not take any arguments.

The `int` and `float` types are strings that can be parsed as integer and floating-point numbers. By default, these types have no limit on the range of acceptable values. However, both types can take a pair of arguments that specify a minimum and maximum range of acceptable values. For example, `int[0, 5]` requires an integer in the range zero to five.

By default, the `string` type does not place any restriction on the length of a string value. However, `string` can take a pair of arguments

Table 9.1: Built-in schema types

| Type | Explanation |
|---|---|
| `boolean` | `"true"` or `"false"` |
| `durationMicroseconds`* | A duration of time |
| `durationMilliseconds`* | A duration of time |
| `durationSeconds`* | A duration of time |
| `enum[name1, ...]` | A enumeration of the specified names |
| `float`* | A decimal number |
| `float_with_units[units1, ...]` | `"<float> <units>"` |
| `int`* | An integer number |
| `int_with_units[units1, ...]` | `"<int> <units>"` |
| `list[type]` | A list of the specified type |
| `memorySizeBytes`* | Memory size expressed as one of: `byte`, `bytes`, `KB`, `MB` or `GB` |
| `memorySizeKB`* | Memory size expressed as one of: `KB`, `MB`, `GB` or `TB` |
| `memorySizeMB`* | Memory size expressed as one of: `MB`, `GB`, `TB` or `PB` |
| `scope` | A scope |
| `string`* | A string |
| `table[name1, type1, ...]` | A table containing columns of the specified names and types |
| `tuple[name1, type1, ...]` | A tuple containing named entries of the specified types |
| `units_with_float[units1, ...]` | `"<units> <float>"` |
| `units_with_int[units1, ...]` | `"<units> <int>"` |

*This type can take an optional `[min, max]` pair of arguments.

that specify a minimum and maximum length for a string. For example, `string[2,5]` requires a string between two and five characters long.

The `enum` type requires one or more arguments. The arguments denote valid values for the `enum` type. For example:

```
String[] schema = new String[] {
    "@typedef colour = enum[grey, white, yellow]",
    ...
};
```

The int_with_units type requires one or more arguments, which specify a enumeration of allowable units. For example, to accept temperature values in the forms "27 Celsius" and "81 Fahrenheit" then you could define a temperature type as follows:

```
String[] schema = new String[] {
    "@typedef temperature = int_with_units[Celsius, Fahrenheit]",
    ...
};
```

As you might expect, the float_with_units type is similar to the int_with_units type, except that the numeric value can be a floating-point number instead of an integer.

The int_with_units and float_with_units types are ideal if the unit is specified *after* the numeric value. If the unit is specified *before* the numeric value then you should use the units_with_int or units_with_float type instead. For example:

```
String[] schema = new String[] {
    "@typedef money = units_with_float[EUR, GBP, USD]",
    ...
};
```

Later, in Section 9.2.5 on page 108, I will explain how to define a schema so that currency symbols (such as €, £ and $) can be used instead of currency names.

The memorySizeBytes, memorySizeKB and memorySizeMB types are built on top of float_with_units. The acceptable units you can use with memorySizeBytes are byte, bytes, KB, MB and GB. The acceptable units for memorySizeKB are KB, MB, GB and TB. And the acceptable units for memorySizeMB are MB, GB, TB and PB. The memory-size types can take a pair of arguments that specify minimum and maximum sizes, but a discussion of this is deferred until Section 9.2.5 on page 108.

The duration types (durationMicroseconds, durationMilliseconds and durationSeconds) are built on top of float_with_units, but they also accept the value "infinite". The acceptable units for use with the duration types are as follows:

**durationMicroseconds:** microsecond, millisecond, second and minute.

**durationMilliseconds:** millisecond, second, minute, hour, day and week.

**durationSeconds:** second, minute, hour, day and week.

You can also specify the plural forms of duration units, for example, `milliseconds` instead of `millisecond`.

The duration types can take a pair of arguments that specify minimum and maximum durations, but a discussion of this is deferred until Section 9.2.5 on page 108.

#### 9.2.4.2 List-based Types

There are three list-based schema types: `list`, `tuple` and `table`. I will discuss each in turn.

The `list` type takes a single argument that denotes the type for every item in the list. For example, the schema below indicates that variable `x` is a list of strings, variable `y` is a list in which each item is an integer, and variable z is a list in which each item is of type `money`:

```
String[] schema = new String[] {
    "@typedef money = units_with_float[EUR, GBP, USD]",
    "x = list[string]",
    "y = list[int]",
    "z = list[money]",
    ...
};
```

The `tuple` type uses a list to emulate a compound data structure, akin to a Pascal `record`, a C/C++ `struct`, or a POJO (that is, a *Plain Old Java Object*). For example, consider the following C++ type:

```
struct person {
    string    name;
    int       age;
    float     height;
};
```

In a configuration file, we might wish to represent `person` data structures as follows:

```
employee = ["John Smith", "42", "186 cm"];
manager = ["Sam White", "39", "170 cm"];
```

Notice that both of the above lists contain three items that correspond to the `name`, `age` and `height` fields of the C++ `struct`. We can validate those lists by using the `tuple` type, which takes one or more *pairs* of arguments that denote the *type* and *name* of a field within the `struct`:

```
String[] schema = new String[] {
```

```
  "@typedef size = float_with_units[cm, m, inches, feet]",
  "@typedef person = tuple[string,name, int,age, size,height]",
  "employee = person",
  "manager = person",
};
```

The arguments to a `tuple` specify not just the *type* of each item in the list, but also the *name* of that item. This enables the schema validator to produce informative error messages. As an example of this, assume that the `example.cfg` file contains the following:

```
foo {
  employee = ["John Smith", "42", "hello"];
  manager = ["Sam White", "39", "170 cm"];
}
```

If we perform a schema validation on the `foo` scope, then we will receive the following error message:

```
example.cfg: bad size value ('hello') for element 3 ('height') of the
'foo.employee' person: should be in the format '<float> <units>' where
<units> is one of: 'cm', 'm', 'inches', 'feet'
```

To understand the `table` type, consider the following example:

```
people = [
  # name           age     height
  #-------------------------------
  "John Smith",   "42",   "186 cm",
  "Sam White",    "39",   "170 cm",
];
```

Syntactically, the `people` variable is a list of strings. However, the list is formatted to look like a table that consists of several rows, each of which contains three columns. The comment at the top indicates the name for each column. A suitable schema for this can be defined with the `table` type, as shown below:

```
String[] schema = new String[] {
  "@typedef size = float_with_units[cm, m, inches, feet]",
  "people = table[string,name, int,age, size,height]"
};
```

The arguments of the `table` type are specified as pairs that indicate the *type* and *name* of each column in the table. If the schema validator encounters an error, then the error message indicates the row and column number of the invalid item, plus the name of the column. For example, if we replace `"186 cm"` in the first row of the example table, then the schema validator will report the following error:

```
example.cfg: bad size value ('hello') for the 'height' column in row 1
of the 'people' table: should be in the format '<float> <units>' where
<units> is one of: 'cm', 'm', 'inches', 'feet'
```

### 9.2.5 Using String-based Arguments

The schema grammar shown in Figure 9.1 on page 100 indicates that an argument used in a rule can be an identifier or a string literal. In the example below, the arguments are identifiers:

```
String[] schema = new String[] {
    "fonts = list[string]",
    "background_colour = enum[grey, white, yellow]",
    "log.level = int[0, 3]"
};
```

It is common to think of identifiers as being textual names, so `string`, `grey`, `white` and `yellow` are clearly identifiers. However, the definition of an identifier given in Section 8.4 on page 83 indicates that numbers are also classified as identifiers. Thus, the arguments `0` and `3` used in the definition of `log.level` are identifiers.

The schema grammar permits string literals to be used (instead of identifiers) for arguments. Thus, the above example could be written in as follows:

```
String[] schema = new String[] {
    "fonts = list[\"string\"]",
    "background_colour = enum[\"grey\", \"white\", \"yellow\"]",
    "log.level = int[\"0\", \"3\"]"
};
```

As you can see, the need to escape the double quotes makes this syntax somewhat cumbersome. For this reason, it is common to write arguments as identifiers rather than as string literals whenever possible. However, sometimes it is *necessary* to write arguments as strings. Two examples come to mind.

First, use of string literals enables schema validation for strings that contain scientific or currency symbols. Thus, the following schema:

```
String[] schema = new String[] {
    "money = units_with_float[\"€\", \"£\", \"$\"]"
};
```

can validate a variable such as:

```
money = "£19.99";
```

Second, you need to use string literals if you want to express minimum and maximum values for memory sizes or durations:

```
String[] schema = new String[] {
    "timeout = durationSeconds[\"10 seconds\", \"5 minutes\"]",
    "RAM_size = memorySizeMB[\"512 MB\", \"4 GB\"]"
};
```

### 9.2.6 Ignore Rules

There are many "framework" libraries that simplify the development of specific types of software, such as GUI applications or client-server applications. Let's assume you are developing a framework library called YAF (an acronym for "Yet Another Framework"). YAF provides useful built-in functionality, but it also has a documented plug-in architecture, so extra functionality can be added easily by third-party companies.

A YAF-based application is likely to require configuration information for all of the following: (1) the core functionality of YAF; (2) each plug-in that is loaded by YAF; and (3) application-level code. Because you are the developer of YAF, you can define a schema for the configuration variables required for (1). However, you are unable to predict what the schema should be for (2) or (3). The *ignore* schema statements, which I will discuss in this section, make it possible for you to write a schema that can validate the configuration information for (1) while ignoring configuration information for (2) and (3). Then, the developer of a plug-in can write another schema to validate the configuration information for that plug-in. Likewise, an application developer can write another schema to validate the configuration information specific to the application code. To illustrate this, consider the example configuration file shown in Figure 9.5, and its schema shown in Figure 9.6.

The `application` and `plugins` scopes store configuration information for application-level code and plugins. The `"@ignoreEverythingIn application"` rule instructs the schema validator to ignore everything (that is, variables and nested scopes) in the `application` scope. The `"@ignoreScopesIn plugins"` rule instructs the schema validator to ignore nested scopes in the `plugins` scope. By *not* ignoring variables in that scope, the schema can validate the `plugins.load` variable.

There is a third *ignore* command called `@ignoreVariablesIn`. That command instructs the schema validator to ignore variables (but not

Figure 9.5: Example configuration for an application built with YAF

```
foo {
    timeout = "2 minutes";
    fonts = ["Times Roman", "Helvetica", "Courier"];
    log {
        dir = "C:\foo\logs";
        level = "1";
    }
    application { ... }
    plugins {
        load = ["tcp", "shared_memory"];
        tcp {
            host = "localhost";
            port = "5050";
            buffer_size = "8 KB";
        }
        ssl { ... }
        shared_memory { ... }
    }
}
```

Figure 9.6: Schema for the configuration shown in Figure 9.5

```
String[] schema = new String[] {
    "timeout = durationMilliseconds",
    "fonts = list[string]",
    "log = scope",
    "log.dir = string",
    "log.level = int[0, 3]",
    "application = scope",
    "@ignoreEverythingIn application",
    "plugins = scope",
    "plugins.load = list[string]",
    "@ignoreScopesIn plugins",
};
```

nested scopes) in the specified scope. That command is provided for completeness, but I have not (yet) found a non-contrived use for it.

If you use the `config2cpp` and `config2j` utilities to generate a schema from a fallback configuration file, then those utilities use built-in heuristics to decide what the schema should be. However, as discussed in Section 6.4 on page 60, you can use a second configuration file, such as

that shown in Figure 6.4 on page 62, to tweak the generated schema. This second configuration file has an `ignore_rules` configuration variable that you can use to specify a list of ignore rules that will then be copied into the generated schema.

## 9.3 Using Code to Define Schema Types

Let's assume you routinely write applications that obtain, say, email addresses and dates from configuration files. Unfortunately, Config4* does not have built-in schema types for email addresses or dates. Because of this, you may decide to write application code to perform the necessary validation checks. Although this approach will work, you will end up cluttering application code with hand-written validation checks. It would be preferable for `emailAddress` and `date` to be built-in types for the schema validator. This raises two interesting questions.

Question one: why doesn't the schema validator have `emailAddress` and `date` types? The answer is a combination of three reasons. First, I have not needed those types in my own projects so far, and I don't have the time to be adding support for types that I (or my colleagues or customers) may not need. Second, there are many different (and sometimes conflicting) standardised ways to write a date, and I don't know which ones I should support in a `date` type. Finally, I realised that I cannot hope to predict all the schema types that somebody, somewhere, will require.

Question two: is it possible to add those types to the schema validator? The answer is yes. The schema validator has an API that enables people to extend it with new schema types. You will find the full details of how to do this in the *Config4* A++ API* and *Config4* Java API* manuals.

## 9.4 Summary

Config4* provides a schema language that you can use to define the entries permitted within the configuration scope for an application. An application can use the `SchemaValidator` class to validate its configuration scope against the schema. If a schema validation error is encountered, then a `ConfigurationException` that indicates the error is thrown.

The schema language is concise and simple to learn. It has many built-in types, plus an API that enables developers to add types specific

to their needs. The schema language also provides several *ignore* rules that enable the entries in a nested scope to be ignored during schema validation. A motivating use of this is to enable schema validation for a framework library to ignore configuration entries specific to plug-ins or application-level code.

# Part IV

# Effective Use of Config4*

# Chapter 10

# Best Practices

## 10.1 Introduction

Much of this manual is concerned on what you *can* do with Config4*. This chapter and the following ones are concerned with what you *should* do. This chapter offers advice on best practices that can help you use Config4* effectively.

## 10.2 Use a Top-level Scope for Each Application

Before you started using Config4*, you probably thought, "My application has a configuration file." Now that you are using Config4*, you should adjust your thinking to be, "My application has a *scope within* a configuration file." By doing this, you make it possible for several applications to share one configuration file. In practice, many people will be content to have a separate configuration file for each application they use. However, it is easy to imagine some people preferring to combine the configuration of several (presumably related) applications in one configuration file.

Some other configuration technologies, such as XML and Java properties files, do not provide good support for one file to contain information for several applications. If your have prior experience with one of these technologies, then it may seem strange at first to confine your application to one scope within a configuration file, but you will soon get used

to it and appreciate its flexibility. At the other extreme is the Windows Registry, which is a configuration source for *all* applications running on a PC. Many people dislike the monolithic nature of the Windows Registry and they may feel wary of sharing one configuration file between multiple applications. I am not advocating that all the applications on a computer *must* share a single configuration file. Rather, I am saying that developers should not force users to adopt a particular granularity of configuration, such as a separate file for each application or one file shared by *all* applications. It is better to leave the choice of granularity to the users.

When using Config4*, it is trivial to design an application so it can obtain its configuration from a scope within a configuration file. To do this, you ensure the application can accept command-line arguments that specify both the configuration file and its scope within that file.

```
myApp.exe -cfg foo.cfg -scope foo
```

Then you use the values obtained from those command-line arguments as parameters when calling `parse()` and the lookup-style operations.

```
cfg = Configuration::create();
cfg.parse(cfgFile);
logDir = cfg.lookupString(scope, "log.dir");
```

It is as simple as that.

By the way, the command-line options do not have to be called `-cfg` and `-scope`; those names are used just as examples. Also, you might prefer to use something other than command-line options, such as environment variables or the Windows Registry, to get the name of the configuration file and the scope within it.

Perhaps an application will default to using its own name as its configuration scope so users are not forced to specify the `-scope` option all the time. If so, then it is important that this is a *default* name for the scope rather than a *hard-coded* name. This is because some users may wish to create several configurations for an application. For example, a user might create scopes called `foo-no-diagnostics` and `foo-with-diagnostics` for an application called Foo.

## 10.3 Naming Convention for Variables

If your application has only a few configuration variables, then you do not need to put much thought into a naming convention for those variables.

However, the number of configuration variables used by your application is likely to increase over time. If your application consists of some core functionality plus optional *plug-ins* (perhaps packaged as UNIX shared libraries, Windows DLLs or Java classes), then you are likely to need configuration variables not just for the core functionality of the application but also for each of its optional plug-ins. For example, I know of a product with plug-ins that has over 600 configuration variables. Most of those configuration variables have sensible default values, so *users* do not need to be concerned with that vast quantity of configuration variables. However, the *developers* of that product had to use a consistent naming scheme to ensure that the names of configuration variables used by one plug-in did not clash with the names of configuration variables used by other plug-ins.

Regardless of whether your application is split over several plug-ins or is packaged as one monolythic executable, your application can be thought of as a collection of logical subsystems. The way to avoid name clashes is to use the name of logical subsystems as prefixes on the names of configuration variables. My preference is to use nested scopes for each logical subsystem within an application, and use the top-level scope for configuration variables that do not neatly fit into any logical subsystem:

```
foo {
    idle_timeout = "5 minutes";
    log {
        file  = "/tmp/fooSrv.log";
        level = "2";
    }
    database {
        host = "...";
        username = "...";
        password = "...";
    }
}
```

Some readers may dislike the indentation caused by nested scopes in the above example. and may prefer a flatter "look and feel" to a configuration file. That is easily achieved by using `"."` (the scoping operator) instead of explicitly opening scopes:

```
foo {
    idle_timeout = "5 minutes";
    log.file  = "/tmp/fooSrv.log";
    log.level = "2";
    database.host = "...";
```

```
    database.username = "...";
    database.password = "...";
}
```

Within a logical subsystem, you should use a consistent convention for compound names, for example, `a_compound_name` or `aCompoundName`.

## 10.4 Fail-fast Configuration

*Fail fast* [Sho04] is a principle of software design. Its essence is that when a problem occurs, an application should fail as soon as possible and in a visible manner, for example, by printing an informative error message. This makes it easier to find and fix bugs (or misconfiguration) which, in turn, leads to more robust applications.

At first sight, the fail-fast principle appears to be in conflict with the use of default (or fallback) configuration values. To see why, consider the following statements.

```
str1 = cfg.lookupString(scope, "log.file");
str2 = cfg.lookupString(scope, "log.file", "foo.log");
```

If the `log.file` variable does not exist, then the first statement throws an exception that contains an informative error message. This is in keeping with the fail-fast principle. In contrast, the second statement returns the specified default value. Returning a default value is appropriate behaviour *if* the configuration variable does not exist. But what if the configuration variable was just misspelt (perhaps as `log.File` instead of `log.file`)? In this case, the misspelling goes undetected, so the fail-fast principle is violated and the application writes to the wrong log file.

Thankfully, the schema validation capabilities of Config4* (provided by the `SchemaValidator` class) can detect misspelt names of configuration variables and so enable you to adhere to the fail-fast principle despite the use of default (or fallback) configuration. You can find an overview of schema validation in Section 3.10 on page 31, and a more complete discussion in Chapter 9.

## 10.5 Zero Configuration

*Zero configuration* is a term used with both software and hardware. It refers to a piece of software or hardware that *is* configurable, but can

work "out of the box" without the need for an explicit configuration step. Zero configuration is prized because it facilitates ease of use.

It is trivial for a Config4*-enabled application to be enabled for zero configuration. This is achieved by the application using *fallback configuration* (Section 3.6.3 on page 25) so the application can work without the need for external configuration. This technique is illustrated by some of the demos, which are discussed in Chapter 11.

When writing the fallback configuration for an application, keep in mind that the fallback configuration can make use of the *adaptive configuration* capabilities of Config4* (Section 2.11 on page 14). In this way, the fallback configuration can take account of the operating system, hostname and username of the person running the application.

## 10.6 Schema Validation for Fallback Configuration

Ideally, you should perform schema validation on *all* sources of configuration information for an application. For example, if you are developing a zero-configuration application, then you should perform schema validation not just on the application's optional configuration file, but also on its fallback configuration. However, this raises a practical problem, as I now discuss.

I explained in Section 9.2.2 on page 102, that the identifier rules in a schema can be either `@optional` or `@required`; if neither is specified, then the default behaviour is `@optional`. The problem is that we need every identifier rule to have *both* the `@optional` and `@required` semantics:

- The zero-configuration nature of the application means that, when performing schema validation on the application's configuration file, all the identifier rules in the schema should have the `@optional` semantics.
- In contrast, when performing schema validation on the application's fallback configuration, all the identifier rules in the schema should have the `@required` semantics.

To support these conflicting requirements, the `validate()` operation of the `SchemaValidator` class can take an optional `forceMode` parameter that "forces" all the identifier rules to have either the `@optional` or `@required` semantics. The use of this parameter is illustrated in Figure 10.1.

Figure 10.1: Specifying a "force mode" for schema validation

```
 1  String schema[] = new String[] {
 2      "idle_timeout = durationMilliseconds",
 3      "log_level = int[0, 5]",
 4      "log_file = string"
 5  };
 6  String cfgFile = ...; // set from a command-line option
 7  String scope = ...;   // set from a command-line option
 8  Configuration cfg = Configuration.create();
 9  try {
10      if (cfgFile != null) {
11          cfg.parse(cfgFile);
12      }
13      cfg.setFallbackConfiguration(Configuration.INPUT_STRING,
14                                   FallbackConfig.getString());
15      SchemaValidator sv = new SchemaValidator();
16      sv.parseSchema(schema);
17      sv.validate(cfg, scope, "", SchemaValidator.FORCE_OPTIONAL);
18      sv.validate(cfg.getFallbackConfiguration(), "", "",
19                  SchemaValidator.FORCE_REQUIRED);
20  } catch(ConfigurationException ex) {
21      System.out.println(ex.getMessage());
22  }
```

Schema validation of the application's configuration file (line 16) uses `FORCE_OPTIONAL` to ensure that all the identifier rules have the `@optional` semantics. (Actually, doing this is unnecessary in the example shown because all the identifier rules in the schema (lines 1–5) have `@optional` semantics by default.) Then, schema validation of the fallback configuration (lines 18–19) uses `FORCE_REQUIRED`. Doing this ensures that an exception will be thrown if the fallback configuration is incomplete. Such an exception will be noticed during application development, which means that the problem of incomplete fallback configuration can be rectified long before the application is shipped to customers or deployed in a production environment.

## 10.7 Working with Lists

Config4* provides operations to lookup a string and convert it to another commonly used type. For example, `lookupInt()` calls `lookupString()`

and tries to convert the returned string to an integer. However, Config4* does *not* provide operations to obtain a list of strings and convert it to a list of other built-in types. For example, Config4* does not provide `lookupListOfInt()` or `lookupListOfBoolean()`. Config4* *could* provide such operations, but this would not be sufficient because some applications would want to lookup a list of mixed types, for example, a list in which the first element is a string, the next element is an integer, the next a duration and so on. Instead, Config4* provides lower-level functionality that enables developers to write their own "lookup a list of exactly what I want" operations.

Perhaps the easiest way to check that a list variable contains exactly what you want is to use the `SchemaValidator` class to validate it. However, Config4* does provide operations that enable you to perform validation checks and data-type conversion on individual elements of a list. It is useful to be aware of the existence of these operations, in case you ever need them. Some operations have names of the form `is<Type>()`, for example, `isBoolean()` and `isInt()`. These operations take a string parameter and return a boolean indicating if the supplied string can be converted to the relevant type. Some other operations have names of the form `stringTo<Type>()`, for example, `stringToInt()` and `stringToBoolean()`. Here is the Java signature for one such operation.

```
int stringToInt(String scope,
                String localName,
                String str) throws(ConfigurationException)
```

This operation tries to convert the specified string (`str`) into an integer. If the conversion fails, then the operation throws an exception containing a descriptive error message. For example, if the `scope` parameter is `"foo"`, the `localName` parameter is `"my_list[3]"` and the configuration file previously parsed was called `example.cfg`, then the message in the exception will be:

```
example.cfg: Non-integer value for 'foo.my_list[3]'
```

The intention is that developers will iterate over all the strings within a list and handcraft the `localName` parameter for each list element to reflect its position within the list: `"my_list[1]"`, `"my_list[2]"`, `"my_list[3]"` and so on. In this way, the `stringTo<Type>()` operations can produce informative exception messages if a data-type conversion fails. Note that although many programming languages, including C++ and Java, index arrays starting from 0, you should format the `localName` parameter so

the index starts at 1. This is to be consistent with the error messages produced by the `SchemaValidator` class.

Lists of mixed types can be used to emulate tables of information. Lists denoting tables are best formatted in the form of a table, with a comment line indicating the meaning of each column.

```
foo {
    price_list = [
      # product     price
      #-------------------------
        "milk",     "$0.99",
        "flour",    "$2.17",
        "jam",      "$1.42"
    ];
}
```

Figure 10.2 contains Java code that calls `lookupList()` to retrive the value of `foo.price_list` and then processes it row by row. The code calls `stringToUnitsWithFloat()` to convert a string, for example, `"$2.17"`, into a `ValueWithUnits` object that provides operations to access the units (`"$"`) and value (`2.17`).

When calling `stringToUnitsWithFloat()`, the code in Figure 10.2 constructs a value for the `localName` parameter that reflects the table element being accessed. If the price column in the first row of the table is changed from `"$0.99"` to `"car"`, then the code in Figure 10.2 will produce the following informative error message:

```
someFile.cfg: invalid price ('car') specified for
'foo.price_list[1].price': should be '<units> <float>'
where <units> are '$', 'EUR'
```

## 10.8 Use a Wrapper Class around Config4*

Perhaps you think Config4* is the best configuration technology in existence and you cannot imagine ever wanting to use anything else. However, there is a good chance that within a few years, or even within a few months, you will switch to something else that will not have the same API as Config4*. This something else might be a different configuration technology, or it might be a newer version of Config4* that has a backwards-incompatible API. When you make such a switch, you will have to modify existing code that uses Config4*. Making such modifications can be time consuming if Config4* is used in many places in your application.

Figure 10.2: Code to process `price_list`

```
String[]          priceList;
String            product;
String            priceStr;
String            localName;
int               i;
int               numRows;
int               row;
final int         numColumns = 2;
ValueWithUnits    vwu;

try {
    priceList = cfg.lookupList(scope, "price_list");
    numRows = priceList.length / numColumns;
    for (i = 0; i < numRows; i++) {
        product   = priceList[i* numColumns + 0];
        priceStr  = priceList[i* numColumns + 1];
        row       = (i / numColumns) + 1;
        vwu       = cfg.stringToUnitsWithFloat(scope,
                          ("price_list["+ row +"].price"),
                          "price", priceStr,
                          new String[]{"$", "EUR"});
        priceCurrency = vwu.getUnits();
        priceAmount   = vwu.getFloatValue();
        process(product, priceCurrency, priceAmount);
    }
} catch(ConfigurationException ex) {
    System.err.println(ex.getMessage());
}
```

You can greatly reduce the impact of moving to a different configuration technology (or upgrading to a newer version of Config4* that has a backwards-incompatible API) by writing your own class that encapsulates the use of Config4*. Some of the demos supplied with Config4* provide examples of such encapsulation. You can find a discussion of the demos in Chapter 11.

## 10.9 Summary

This chapter has provided advice on best practices for using Config4*. Following the advice will help you to develop applications that are flex-

ible, user-friendly and easy to maintain.

# Chapter 11

# Demonstration Applications

## 11.1 Introduction

Several demonstration applications are supplied with implementations of Config4*. You can find these in the `demo` directory hierarchy of your distribution. I advise readers to examine the source code of these demos while reading the discussion of the demos in this chapter.

## 11.2 The `simple-encapsulation` Demo

The `simple-encapsulation` demo defines a class that provides the configuration capabilities for a hypothetical *Foo* application. This application requires only seven configuration variables so, rather than expose general purpose lookup-style operations, the `FooConfiguration` class provides accessor operations for these seven pieces of configuration. The class surrounds calls to Config4* with a `try-catch` clause so the information inside a Config4* exception can be copied to an application-specific exception that is thrown. In these simple ways, the `FooConfiguration` class encapsulates the use of Config4*, thus making it easy for a maintainer of the *Foo* application to migrate to using a different configuration technology, should the need ever arise.

Despite `FooConfiguration` having a simple-to-use API, the class encapsulates some powerful features of Config4*. First, the class allows

the default security policy to be overridden (Section 5.4 on page 54). Second, the class uses *fallback configuration* (Section 3.6.3 on page 25) so users can run the Foo application without a configuration file. The source used for the fallback configuration is an embedded string that the `Makefile` or `build.xml` file generates by use of the `config2cpp` or `config2j` utility.

The `FooConfiguration::parse()` operation takes four string parameters that specify the configuration source (a file or `"exec#..."`) and scope within it, and a security source and scope within it. Empty strings can be passed for any or all of these parameters, thus making a user-specified configuration source and user-specified security policy optional.

If `FooConfiguration` were re-implemented to use, say, an XML file or a Java properties file then the four parameters to `parse()` would likely be replaced by a single parameter denoting the name of the XML file or Java properties file. This change in the public API of `FooConfiguration` is likely to have minimal knock-on effects in the rest of the Foo application. In particular, the parsing of command-line arguments and subsequent creation of a `FooConfiguration` object are the *only* pieces of code likely to be affected.

## 11.3 The `encapsulate-lookup-api` Demo

The `encapsulate-lookup-api` demo defines a class that provides the configuration capabilities for a hypothetical *Foo* application. This application requires general purpose lookup-style operations when accessing configuration information. The `FooConfiguration` class provides its own lookup-style operations that internally delegate to the lookup operations of Config4*. In this way, the `FooConfiguration` class encapsulates the use of Config4*, thus making it easy for a maintainer of the *Foo* application to migrate to using a different configuration technology, should the need ever arise.

The delegation logic is straightforward but a bit verbose due to the need to surround calls to Config4* with a `try-catch` clause so the information inside a Config4* exception can be copied to an application-specific exception that is thrown. This type of delegation logic occupies only a few lines of code for each lookup-style operation. The verbosity arises because Config4* provides *so many* lookup-style operations. Config4* has lookup operations for many different types, and for each type the lookup operation is overloaded to provide a "lookup with a default

value" and a "lookup without a default value." To keep the volume of code manageable, the demo does *not* provide the "lookup with a default value" version of operations. Instead, it uses *fallback configuration* (Section 3.6.3 on page 25) to provide default values.

## 11.4 The `log-level` Demo

Many applications have the ability to print diagnostic messages (as a troubleshooting aid when something goes wrong), and use a command-line option or variable in a configuration file to set the diagnostics level. A primitive way to control the diagnostics level is to have a, say, a `"-d <int>"` command-line option that sets the diagnostics level for the entire application. However, this simplistic approach can result in "too many" irrelevant diagnostics messages being printed, which can hinder attempts to diagnose a problem.

A better approach is to provide an independent diagnostic level for each component in an application. This makes it possible to selectively turn on diagnostics for some components, while turning off diagnostics for other components. This flexible approach is becoming increasingly common, though the granularity of a "component" varies widely. In some applications, a component might be coarse-grained, such as an entire functional subsystem or a plug-in. In other applications, a component might be finer-grained, such as individual classes (which is common in Log4J-based applications), or even individual operations on a class.

If you want to use this "separate log-level for each component" technique in a Config4*-based application, then you might think of using a separate configuration variable for each component. For example:

```
log_level {
    component1 = "2";
    component2 = "0";
    component3 = "0";
};
```

However, that approach does not scale well: if you have hundreds of components in your application, then you will need hundreds of configuration variables to control their log levels.

My preferred approach is to use a two-column table that provides a mapping from the wildcarded name of a component to a log level. You can see an example of this in Figure 11.1. The string `"A::op3"`

denotes operation `op3()` in the class `A`.[1] The wildcard character (`"*"`) matches zero or more characters, and it might be used to match, say, the names of all operations within a specific class (`"B::*"`), all create-style operations, regardless of the class in which they appear (`"*::create*"`), or all operations in all classes (`"*"`).

Figure 11.1: Using a table to specify log levels

```
log_level = [
    # wildcarded component name   log level
    #-------------------------------------
    "A::op3",                     "4",
    "B::op1",                     "3",
    "B::*",                       "1",
    "*",                          "0",
];
```

When a component needs to determine its log level, it iterates through the rows of the table, and uses the log level of the first matching wildcarded entry.[2] Thus, the last line of the table can specify a default log level (by using `"*"` as the wildcarded component name), and earlier lines in the table can specify a different log level for individual components (or groups of components). This combination of wildcarding and defaulting means that the table can remain short even if an application contains hundreds or thousands of components.

The `log-level` demo provides code that illustrates how to use a table to specify wildcarded log levels.

## 11.5 The `recipes` Demo

The `recipes` demo provides an example of how to process scopes and variables that contain the `"uid-"` prefix. The `recipes.cfg` file contains a collection of `uid-recipe` scopes, like those shown in Figure 11.2.

The `RecipeFileParser` class provides a simple API for parsing such a file and iterating over the recipes contained within it. The `parse()` operation within this class illustrates how to perform schema validation for scopes and variables that contain the `"uid-"` prefix. In addition, the

[1] C++ uses `"::"` as the scoping operator. For a Java application, an entry in the first column of the table might be written as `"com.example.mypackage.A.op3"`.

[2] Config4* provides a `patternMatch()` operation that an application can use for this purpose.

Figure 11.2: File of recipes

```
uid-recipe {
    name = "Tea";
    ingredients = ["1 tea bag", "cold water", "milk"];
    uid-step = "Pour cold water into the kettle";
    uid-step = "Turn on the kettle";
    uid-step = "Wait for the kettle to boil";
    uid-step = "Pour boiled water into a cup";
    uid-step = "Add tea bag to cup & leave for 3 minutes";
    uid-step = "Remove tea bag";
    uid-step = "Add a splash of milk if you want";
}
uid-recipe {
    name = "Toast";
    ingredients = ["Two slices of bread", "butter"];
    uid-step = "Place bread in a toaster and turn on";
    uid-step = "Wait for toaster to pop out the bread";
    uid-step = "Remove bread from toaster and butter it";
}
```

`parse()` and `getRecipeSteps()` operations illustrate how to use pass a *filter* string such as `"uid-recipe"` or `"uid-step"` to `listFullyScopedNames()` and `listLocallyScopedNames()` to get a list of `"uid-"` entries.

## 11.6 The `extended-schema-validator` Demo

This demo application illustrates how to enhance the Config4* schema validator with knowledge of additional schema types.

The `ExtendedSchemaValidator` class illustrates how to write a subclass of `SchemaValidator`. Most of the code in this class is boilerplate text that delegates to the parent class. The important part of this class is the implementation of `registerTypes()`, which registers a singleton instance of each new schema type. For the purposes of this demo, this operation registers the `SchemaTypeHex` class, which performs schema validation for hexadecimal numbers.

A class that provides a new schema type must implement two operations:

**`checkRule()`** is invoked when the schema type is used in a schema rule.

**`isA()`** is invoked during schema validation of a configuration file.

The `SchemaTypeHex` class illustrates how to implement the above operations. In addition, the class provides some utility functions that might be useful to application developers: `lookupHex()`, `stringToHex()` and `isHex()`.

The `FooConfiguration` class encapsulates use of Config4* and the new schema type. This class illustrates how to make use of the enhanced schema validator and the utility functions provided by the `SchemaTypeHex` class.

## 11.7 Summary

This chapter has discussed the demonstration applications supplied with Config4*. To fully understand the information here, you should examine the source code of the demonstration applications while reading this chapter.

If you want more in-depth advice on how to exploit the full potential of Config4*, then you might wish to read the *Config4* Practical Usage Guide*.

# Bibliography

[Ray03] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional Computing Series, 2003.

[Sho04] Jim Shore. Fail fast. *IEEE Software*, pages 21–25, Sept/Oct 2004. www.martinfowler.com/ieeeSoftware/failFast.pdf.

[Wal02] Priscilla Walmsley. *Definitive XML Schema*. Prentice Hall, 2002.