Java API 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 Purpose of this Manual

This manual provides a reference for the application programming interface (API) of Config4J. This manual does *not* provide a beginner's tutorial on Config4J. You can find such a tutorial in the *Config4* Getting Started Guide*.

The rest of this chapter discusses the principles that underpin the API of Config4J. Knowledge of these principles makes it easier to understand the API. The chapters that follow discuss the API of individual classes.

1.2 Package

All the classes of Config4J are defined in the `org.config4j` package. For conciseness, the package prefix is not explicitly stated in the discussion of classes and operations in this manual.

1.3 Portability

One of the design goals of Config4J is portability. Of course, Java source code (and byte code) is portable across many operating systems, so that aspect of portability is obtained trivially. However, another aspect of the portability goal is that Config4J should compile with both new and older Java compilers. For this reason, Config4J avoids using relatively new language features. In particular, Config4J avoids language features

introduced in Java 5: annotations, generics and enumerations. Config4J also avoids use of the `assert` keyword, which was introduced in Java 1.4. You should be able to use Config4J with Java 1.3 or newer.[1] Config4J may even work with Java 1.2, but I have not tried that.

1.4 Error Reporting

The Config4J parser does not make any attempt at error recovery. Instead, it stops at the first error it encounters, and reports the error by throwing an exception. The lack of error recovery helps to keep the implementation of the parser simple. It also simplifies the public API because the ability to report multiple parsing errors would have required a more complex API.

1.5 Specifying Scoped Names

Many of the operations in Config4J work with scoped names, for example, `foo_srv.log.dir`. Typically, the first part, `foo_srv`, is a scope for a particular application, and the remainder, `log.dir`, is a configuration variable used by that application. It can be useful to change the application name (`foo_srv`) *without* needing to change lots of code. It would not be possible to do this if `foo_srv.log.dir` appeared in application code. Instead, it is best for the two parts of a name to be specified separately, and then merged to form the fully-scoped name.

It would be tedious for developers to do such merging manually. To avoid this, the Config4J operations that work on scoped names take *two* string parameters. Internally, the operations merge the strings to obtain a fully-scoped name. For example, you can access the value of `foo_srv.log.dir` with the following statement:

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

The intention is that an application can declare a variable called, say, `scope` and initialize its value from a command-line argument. Then the application can access configuration information from within that scope by using code like that shown below:

[1] Unfortunately, I do not have a Java 1.3 compiler installed on my development machine. Instead, I compile Config4J with `javac -source 1.3`, which would complain if I tried to use newer language features. However, that is not a foolproof way to ensure compatibility with Java 1.3 because I might be accidentally using a class or operation that was introduced in a later version of Java.

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

By rerunning an application with a different command-line argument, you can change the scope used to configure the application. This provides a lot of flexibility. 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 instance of a replicated server application.

Chapter 2

The `Configuration` Class

2.1 The `ConfigurationException` Class

A runtime exception of type `ConfigurationException` is thrown if any Config4J operation fails. The public API of this class is shown in Figure 2.1.

Figure 2.1: The `ConfigurationException` class

```
package org.config4j;

public class ConfigurationException extends java.lang.RuntimeException
{
    public ConfigurationException(String msg);
}
```

2.2 The `create()` Operation

Most of the public API of Config4J is defined in the `Configuration` class. This is an *abstract* class, so you cannot create an instance of it directly. Instead, you create a `Configuration` object by calling the static `create()` operation, which is shown in Figure 2.2.

A newly created `Configuration` object is empty initially. You can then populate it and access its contents, as I will discuss in the following sections of this chapter.

Figure 2.2: Initialization API for `Configuration`

```
package org.config4j;

public abstract class Configuration
{
    public static Configuration create();
    ...
}
```

Most of the operations defined in the `Configuration` class can throw `ConfigurationException` exceptions, so those operations should be called from inside a `try-catch` clause. However, the `create()` operation does *not* throw that exception, so, as shown in Figure 2.3, it can be called from outside a `try-catch` clause.

Figure 2.3: Example of creating a `Configuration` object

```
import org.config4j.Configuration;
import org.config4j.ConfigurationException;
...
Configuration * cfg = Configuration.create();
try {
    ... // invoke operations on cfg
} catch(ConfigurationException ex) {
  System.err.println(ex.getMessage());
}
```

2.3 Utility Operations

Config4J provides several utility operations, shown in Figure 2.4, that you may need to use from time to time.

As I discussed in Section 1.5 on page 2, many Config4* operations take a pair of parameters, `scope` and `localName`, that, when merged, specify the fully-scoped name of an entry in a `Configuration` object. The `mergeNames()` operation performs that merging. The fully-scoped name is usually of the form *scope.localName*, but if either `scope` or `localName` is an empty string, then the dot (`"."`) is omitted when performing the merge.

The `patternMatch()` operation compares a string against a pattern,

Figure 2.4: Utility operations

```
package org.config4j;

public abstract class Configuration
{
    public static String mergeNames(String scope, String localName);
    public static boolean patternMatch(String str, String pattern);
    public String getenv(String name);
    ...
}
```

and returns `true` if they match. Within the pattern, the `"*"` character acts as a wildcard that matches zero or more characters. For example:

```
Configuration.patternMatch("Hello, world", "Hello*") → true
Configuration.patternMatch("Hello, world", "123*89") → false
```

The `getenv()` operation returns the value of the specified environment variable if it exists, and `null` otherwise.

2.4 The `parse()`, `fileName()` and `empty()` Operations

Figure 2.5 shows the signatures of the `fileName()`, `parse()` and `empty()` operations.

The `fileName()` operation returns the name of the most recently parsed file. If `parse()` has not previously been called, then `fileName()` returns an empty string.

I defer discussion of the one-parameter version of `parse()` until Section 2.4.4 on page 10 because it is just a simplified version of the three-parameter version of `parse()`. In the three-parameter version of `parse()`, the value of the first parameter determines the meaning of the other parameters.

- If the first parameter is `INPUT_FILE`, then the second parameter is the name of the file to be parsed, and the third parameter is ignored.

  The second parameter will be the value returned from future calls to `fileName()`.

Figure 2.5: The `parse()`, `fileName()` and `empty()` operations

```
package org.config4j;

public abstract class Configuration
{
    // Constant values used for sourceType in parse()
    public static final int INPUT_FILE   = 1;
    public static final int INPUT_STRING = 2;
    public static final int INPUT_EXEC   = 3;

    public String fileName();
    public void parse(
        int          sourceType,
        String       source,
        String       sourceDescription) throws ConfigurationException;
    public void parse(
        int          sourceType,
        String       source) throws ConfigurationException;
    public void parse(String sourceTypeAndSource)
                                        throws ConfigurationException;
    public void empty();
    ...
}
```

- If the first parameter is `INPUT_STRING`, then the second parameter is a string to be parsed.

  If the third parameter is *not* an empty string, then it will be the value returned from future calls to `fileName()`; otherwise the string `"<string-based configuration>"` will be the value returned from future calls to `fileName()`.

- If the first parameter is `INPUT_EXEC`, then the second parameter is an external command to be executed and whose standard output is to be parsed.

  If the third parameter is *not* an empty string, then it will be the value returned from future calls to `fileName()`; otherwise the string resulting from appending the second parameter to `"exec#"` will be the value returned from future calls to `fileName()`.

Calling the two-parameter version of `parse()` is similar calling the three-parameter version with an empty string for the last parameter.

The string returned from `fileName()` is used at the start of text messages inside exceptions. For example, many components of Config4J (including the parser, schema validator and lookup operations) format exception messages as shown below:

```
if (...) {
    throw ConfigurationException(cfg->fileName()
                                 + ": something went wrong");
}
```

For this reason, if you call `parse()` with `INPUT_STRING` for the first parameter, then you should ensure that the value of the third parameter acts as a descriptive "file name".

2.4.1 Parsing a File

The code segment in Figure 2.6 shows an example use of `parse()`. The `create()` operation creates a `Configuration` object that is empty initially. Then the `parse()` operation is used to populate the `Configuration` object. A `try-catch` clause is used to print any exception that might be thrown. Once the `Configuration` object has been populated, lookup operations (which I will discuss in Section 2.6 on page 13) can be used to access information in it.

Figure 2.6: An example of using `parse()`

```
Configuration cfg = Configuration.create();
try {
    cfg.parse(Configuration.INPUT_FILE, "myFile.cfg");
    ... // invoke lookup operations
} catch(ConfigurationException ex) {
    System.err.println(ex.getMessage());
}
```

2.4.2 Parsing the Output of a Command

The example in Figure 2.6 used the following to parse a file:

```
cfg.parse(Configuration.INPUT_FILE, "myFile.cfg");
```

If, instead of parsing a file, you want to execute a command and parse its standard output, then you can do so as follows:

```
cfg.parse(Configuration.INPUT_EXEC, "curl -sS http://host/file.cfg");
```

Using INPUT_EXEC for the first parameter tells parse() to to interpret the second parameter as the name of a command to be executed.

2.4.3 Parsing a String

The example in Figure 2.6 used the following to parse a file:

```
cfg.parse(Configuration.INPUT_FILE, "myFile.cfg");
```

If, instead of parsing a file, you want to parse a string, then you can do so as follows:

```
String cfgStr = ...;
cfg.parse(Configuration.INPUT_STRING, cfgStr, "embedded configuration");
```

Using INPUT_STRING for the first parameter tells parse() to interpret the second parameter as configuration data that should be parsed directly, and the third parameter is the "file name" that will be used when reporting errors. You can initialise the second parameter in a variety of ways, for example:

- You could use the config2cpp utility to convert a configuration file into (a class wrapper around) a string that is embedded into the application. In this case, you might use "embedded configuration" or "fallback configuration" as the third parameter.
- Perhaps you are developing a client-server application that serialises messages into Config4* syntax and then transmits them across a socket connection. In the receiving application, the second parameter to parse() would be a message that was read from a socket connection. In this case, you might use "incoming message" as the third parameter.

2.4.4 The Simplified Version of parse()

The one-parameter version of parse() is a simplification wrapper around the three-parameter version. Its implementation is shown in Figure 2.7.

The following examples show how to use this simplified version:

```
cfg->parse("exec#curl -sS http://host/file.cfg");
cfg->parse("file#file.cfg");
cfg->parse("file.cfg");
```

Figure 2.7: Simplified version of `parse()`

```
public void parse(String sourceTypeAndSource)
                                        throws ConfigurationException
{
    int       cfgSourceType;
    String    cfgSource;

    if (sourceTypeAndSource.startsWith("exec#")) {
        cfgSource = sourceTypeAndSource.substring(5);
        cfgSourceType = Configuration.INPUT_EXEC;
    } else if (sourceTypeAndSource.startsWith("file#")) {
        cfgSource = sourceTypeAndSource.substring(5);
        cfgSourceType = Configuration.INPUT_FILE;
    } else {
        cfgSource = sourceTypeAndSource;
        cfgSourceType = Configuration.INPUT_FILE;
    }
    parse(cfgSourceType, cfgSource);
}
```

In practice, the parameter to this operation is unlikely to be hard-coded into an application, but rather will come from, say, a command-line option or an environment variable.

2.4.5 Parsing Multiple Files and the `empty()` Operation

If you want to parse multiple configuration files, then you can use multiple `Configuration` objects. Alternatively, you can reuse the same object multiple times. If you do this, then you will probably want to call `empty()` between successive calls of `parse()`, as shown in Figure 2.8. The `empty()` operation has the effect of removing all variables and scopes from the `Configuration` object.

It is legal to call `parse()` multiple times *without* calling `empty()` between successive calls. If you do this, then each call to `parse()` merges its information with information already in the `Configuration` object. The Config4* parser implements the `@include` statement by (recursively) calling `parse()`, so you can think of multiple calls to `parse()` without calls to `empty()` as being similar to multiple `@include` statements.

It is difficult to think of a compelling reason why you might want to use a single `Configuration` object to parse multiple configuration files

Figure 2.8: Calling `parse()` and `empty()` multiple times

```
Configuration cfg = Configuration.create();
try {
    cfg.parse("file1.cfg");
    ... // Access the configuration information
    cfg.empty();
    cfg.parse("file2.cfg");
    ... // Access the configuration information
    cfg.empty();
    cfg.parse("file3.cfg");
    ... // Access the configuration information
    cfg.empty();
    cfg.parse("file4.cfg");
    ... // Access the configuration information
} catch(ConfigurationException ex) {
    System.err.println(ex.getMessage()):
}
```

without calling `empty()` between successive calls of `parse()`. However, it is useful to know what the semantics of doing so are, because it can help you understand what is happening if you forget to call `empty()` between calls to `parse()` on the same `Configuration` object.

2.5 Insertion and Removal Operations

Most applications will populate a `Configuration` object by parsing a configuration file. However, it is possible to populate a `Configuration` object by using the operations shown in Figure 2.9.

The `insertString()` and `insertList()` operations add a *name=value* entry using the fully-scoped name (obtained by merging the `scope` and `localName` parameters) and the specified value. If a variable of the same name already exists in the `Configuration` object, then it is replaced with the new value.

The `ensureScopeExists()` operation merges the `scope` and `localName` parameters to obtain a fully-scoped name. It ensures that a scope with this fully-scoped name exists. If any ancestors of the specified scopes are missing, then they are also created. Internally, the `insertString()` and `insertList()` operations call `ensureScopeExists()`. Because of this, applications rarely need to call `ensureScopeExists()` directly.

The `remove()` operation merges `scope` and `localName` to form a fully-

Figure 2.9: Insertion and removal operations

```
package org.config4j;

public abstract class Configuration
{
    public void insertString(
        String    scope,
        String    localName,
        String    strValue) throws ConfigurationException;
    public void insertList(
        String    scope,
        String    localName,
        String[]  listValue) throws ConfigurationException;
    public void ensureScopeExists(
        String    scope,
        String    localName) throws ConfigurationException;
    public void remove(
        String    scope,
        String    localName) throws ConfigurationException;
    ...
}
```

scoped name. It then removes the entry with the specified name.

If you are making use of identifiers that have `"uid-"` prefixes, then it is your duty to expand such identifiers before invoking any of the operations listed in Figure 2.9. Section 2.12 on page 29 explains how to expand identifiers that have `"uid-"` prefixes.

2.6 The `lookup<Type>()` Operations

Figure 2.10 lists lookup-style operations that you can use to access the values of configuration variables. There are a *lot* of lookup operations, for the following reasons.

- Syntactically, variables in a configuration file are either strings or lists. However, strings are often used to encode other types, such as integers, floats, booleans, durations and so on. Because of this, there are type-safe lookup operations that convert a string value to another format. For example, `lookupInt()` converts a string value to an `int`, and `lookupBoolean()` converts a string value to a `bool`.

- The lookup operations are overloaded so that you can optionally specify a default value that should be returned if the specified configuration variable is not present.

The lookup operations perform error checking. For example, if the `lookupInt()` operation cannot convert the string value into an integer, then it throws an exception that contains an easy-to-understand error message. Likewise, if you do not specify a default value to a lookup operation and the specified configuration variable is missing (from both the main configuration object and fallback configuration), then an exception is thrown.

2.6.1 Lookup Operations for Enumerated Types

Among other parameters, the `lookupEnum()` operation takes an array of `EnumNameAndValue` objects. This operation calls `lookupString()` and then uses the array to convert the string value into an integer. For example:

```
EnumNameAndValue colourInfo[] = new EnumNameAndValue[] {
    new EnumNameAndValue("red",   0),
    new EnumNameAndValue("green", 1),
    new EnumNameAndValue("blue",  2)
};
colour = cfg.lookupEnum(scope, "font_colour", "colour", colourInfo);
```

The `typeName` parameter (`"colour"` in the above example) specifies the "type name" of the enum names, and is used to construct an informative error message if an exception is thrown.

2.6.2 Lookup Operations for Unit-based Types

Lookup operations that have `Units` in their name take, among other parameters, an array of strings that specifies the allowed units. An example can be seen in Figure 2.11. The `typeName` parameter (`"price"` in the example) specifies the "type name" of correctly-formatted strings, and is used to construct an informative error message if an exception is thrown.

2.7 The `type()` and `is<Type>()` Operations

Figure 2.12 shows the operations you can use to query type information.

Figure 2.10: The lookup<Type>() operations

```
package org.config4j;

public class EnumNameAndValue
{
    public EnumNameAndValue(String name, int value);
    public String getName();
    public int    getValue();
}

public class ValueWithUnits
{
    public ValueWithUnits();
    public ValueWithUnits(int value, String units);
    public ValueWithUnits(float value, String units);
    public int    getIntValue();
    public float  getFloatValue();
    public String getUnits();
}

public abstract class Configuration
{
    public String lookupString(
        String        scope,
        String        localName,
        String        defaultVal) throws ConfigurationException;
    public String lookupString(String scope, String localName)
                                        throws ConfigurationException;
    public String lookupString(
        String        scope,
        String        localName,
        String        defaultVal) throws ConfigurationException;
    public String lookupString(String scope, String localName)
                                        throws ConfigurationException;
    public String[] lookupList(
        String        scope,
        String        localName,
        String[]      defaultArray) throws ConfigurationException;
    public String[] lookupList(String scope, String localName)
                                        throws ConfigurationException;
... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    public int lookupInt(
        String        scope,
        String        localName,
        int           defaultVal) throws ConfigurationException;
    public int lookupInt(String scope, String localName)
                                      throws ConfigurationException;

    public float lookupFloat(
        String        scope,
        String        localName,
        float         defaultVal) throws ConfigurationException;
    public float lookupFloat(String scope, String localName)
                                      throws ConfigurationException;

    public int lookupEnum(
        String              scope,
        String              localName,
        String              typeName,
        EnumNameAndValue[]  enumInfo,
        String              defaultVal) throws ConfigurationException;
    public int lookupEnum(
        String              scope,
        String              localName,
        String              typeName,
        EnumNameAndValue[]  enumInfo,
        int                 defaultVal) throws ConfigurationException;
    public int lookupEnum(
        String              scope,
        String              localName,
        String              typeName,
        EnumNameAndValue[]  enumInfo) throws ConfigurationException;

    public boolean lookupBoolean(
        String              scope,
        String              localName,
        boolean             defaultVal) throws ConfigurationException;
    public boolean lookupBoolean(
        String              scope,
        String              localName) throws ConfigurationException;
... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    public ValueWithUnits lookupFloatWithUnits(
        String              scope,
        String              localName,
        String              typeName,
        String[]            allowedUnits)
                                    throws ConfigurationException;
    public ValueWithUnits lookupFloatWithUnits(
        String              scope,
        String              localName,
        String              typeName,
        String[]            allowedUnits,
        ValueWithUnits      defaultValueWithUnits)
                                    throws ConfigurationException;

    public ValueWithUnits lookupUnitsWithFloat(
        String            scope,
        String            localName,
        String            typeName,
        String[]          allowedUnits) throws ConfigurationException;
    public ValueWithUnits lookupUnitsWithFloat(
        String            scope,
        String            localName,
        String            typeName,
        String[]          allowedUnits,
        ValueWithUnits    defaultValueWithUnits)
                                    throws ConfigurationException;

    public ValueWithUnits lookupIntWithUnits(
        String            scope,
        String            localName,
        String            typeName,
        String[]          allowedUnits) throws ConfigurationException;
    public ValueWithUnits lookupIntWithUnits(
        String            scope,
        String            localName,
        String            typeName,
        String[]          allowedUnits,
        ValueWithUnits    defaultValueWithUnits)
                                    throws ConfigurationException;
... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    public ValueWithUnits lookupUnitsWithInt(
        String            scope,
        String            localName,
        String            typeName,
        String[]          allowedUnits) throws ConfigurationException;
    public ValueWithUnits lookupUnitsWithInt(
        String            scope,
        String            localName,
        String            typeName,
        String[]          allowedUnits,
        ValueWithUnits    defaultValueWithUnits)
                                      throws ConfigurationException;
    public int lookupDurationMicroseconds(
        String            scope,
        String            localName,
        int               defaultVal) throws ConfigurationException;
    public int lookupDurationMicroseconds(
        String            scope,
        String            localName) throws ConfigurationException;
    public int lookupDurationMilliseconds(
        String            scope,
        String            localName,
        int               defaultVal) throws ConfigurationException;
    public int lookupDurationMilliseconds(
        String            scope,
        String            localName) throws ConfigurationException;
    public int lookupDurationSeconds(
        String            scope,
        String            localName,
        int               defaultVal) throws ConfigurationException;
    public int lookupDurationSeconds(
        String            scope,
        String            localName) throws ConfigurationException;
    public int lookupMemorySizeBytes(
        String            scope,
        String            localName,
        int               defaultVal) throws ConfigurationException;
    public int lookupMemorySizeBytes(
        String            scope,
        String            localName) throws ConfigurationException;
... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    public int lookupMemorySizeKB(
        String            scope,
        String            localName,
        int               defaultVal) throws ConfigurationException;
    public int lookupMemorySizeKB(
        String            scope,
        String            localName) throws ConfigurationException;
    public int lookupMemorySizeMB(
        String            scope,
        String            localName,
        int               defaultVal) throws ConfigurationException;
    public int lookupMemorySizeMB(
        String            scope,
        String            localName) throws ConfigurationException;
    public void lookupScope(
        String            scope,
        String            localName) throws ConfigurationException;
    ...
}
```

Figure 2.11: Example invocation of lookupUnitsWithFloat

```
String currencies[] = new String[] {"£", "$", "€"};
ValueWithUnits vu = cfg.lookupUnitsWithFloat(scope, "discount_price",
                                             "price", currencies);
System.out.println("Currency = " + vu.getUnits());
System.out.println("Amount   = " + vu.getFloatValue());
```

The type() operation merges the scope and localName parameters to form the fully-scoped name of an entry in the Configuration object, and then returns the type of that entry. The return value of this operation will be one of the following:

Return value	Meaning
Configuration.CFG_NO_VALUE	The entry does not exist
Configuration.CFG_STRING	The entry is a string variable
Configuration.CFG_LIST	The entry is a list variable
Configuration.CFG_SCOPE	The entry is a scope

Operations with names of the form is<Type>() return true if the parameter is of the specified type. For example:

Figure 2.12: The type() and is<Type>() operations

```
package org.config4j;
public class EnumNameAndValue {
    public EnumNameAndValue(String name, int value);
    public String getName();
    public int    getValue();
}
public abstract class Configuration {
    // Type constants
    public static final int CFG_NO_VALUE      = 0;// bit masks
    public static final int CFG_STRING        = 1;// 0001
    public static final int CFG_LIST          = 2;// 0010
    public static final int CFG_SCOPE         = 4;// 0100
    public static final int CFG_VARIABLES     = 3;// STRING|LIST
    public static final int CFG_SCOPE_AND_VARS= 7;// STRING|LIST|SCOPE

    public int type(String scope, String localName);
    public boolean isBoolean(String str);
    public boolean isInt(String str);
    public boolean isFloat(String str);
    public boolean isDurationMicroseconds(String str);
    public boolean isDurationMilliseconds(String str);
    public boolean isDurationSeconds(String str);
    public boolean isMemorySizeBytes(String str);
    public boolean isMemorySizeKB(String str);
    public boolean isMemorySizeMB(String str);
    public boolean isEnum(
        String               str,
        EnumNameAndValue[]   enumInfo);
    public boolean isFloatWithUnits(
        String               str,
        String[]             allowedUnits);
    public boolean isIntWithUnits(
        String               str,
        String[]             allowedUnits);
    public boolean isUnitsWithFloat(
        String               str,
        String[]             allowedUnits);
    public boolean isUnitsWithInt(
        String               str,
        String[]             allowedUnits);
    ...
}
```

```
cfg.isBoolean("true")                    → true
cfg.isBoolean("Fred")                    → false
cfg.isDurationSeconds("2.5 minutes")     → true
cfg.isDurationSeconds("100 milliseconds") → false
```

The `isEnum()` operation takes two parameters—a string to be tested and an array of `EnumNameAndValue` objects—as you can see in Figure 2.13.

Figure 2.13: Examples of calling `isEnum()`

```
EnumNameAndValue colourInfo[] = new EnumNameAndValue[] {
    new EnumNameAndValue("red",   0),
    new EnumNameAndValue("green", 1),
    new EnumNameAndValue("blue",  2)
};
cfg.isEnum("red", colourInfo) → true
cfg.isEnum("foo", colourInfo) → false
```

The `is<Type>()` operations with `"Units"` in their name take two parameters—a string to be tested and an array of strings that specifies the allowed units—as you can see in Figure 2.14.

Figure 2.14: Examples of calling `isUnitsWithFloat`

```
String currencies[] = new String[] {"£", "$", "€"};
cfg.isUnitsWithFloat("£19.99", currencies) → true
cfg.isUnitsWithFloat("foobar", currencies) → false
```

2.8 The `stringTo<Type>()` Operations

Figure 2.15 lists operations that can convert a string value to another type.

An operation with a name of the form `stringTo<Type>()` converts a string into the specified type. If the conversion fails, then the operation throws an exception containing an informative error message. The error message will indicate that the problem arose with the variable identified by the fully-scoped name (obtained by merging the `scope` and `localName` parameters) in the `fileName()` configuration file.

As an example, consider a call to `stringToInt()` in which the `scope` parameter is `"foo"`, the `localName` parameter is `"my_list[3]"` and the `str`

Figure 2.15: The stringTo<Type>() operations

```
package org.config4j;

public class EnumNameAndValue {
    public EnumNameAndValue(String name, int value);
    public String getName();
    public int    getValue();
}

public class ValueWithUnits {
    public ValueWithUnits();
    public ValueWithUnits(int value, String units);
    public ValueWithUnits(float value, String units);
    public int    getIntValue();
    public float  getFloatValue();
    public String getUnits();
}

public abstract class Configuration
{
    public boolean stringToBoolean(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
    public int stringToInt(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
    public float stringToFloat(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
    public int stringToDurationSeconds(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
    public int stringToDurationMilliseconds(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
... continued on the next page
```

Figure 2.15 (continued): The stringTo<Type>() operations

```
... continued from the previous page
    public int stringToDurationMicroseconds(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
    public int stringToMemorySizeBytes(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
    public int stringToMemorySizeKB(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;
    public int stringToMemorySizeMB(
        String            scope,
        String            localName,
        String            str) throws ConfigurationException;

    public int stringToEnum(
        String              scope,
        String              localName,
        String              type,
        String              str,
        EnumNameAndValue[]  enumInfo) throws ConfigurationException;

    public ValueWithUnits stringToFloatWithUnits(
        String              scope,
        String              localName,
        String              typeName,
        String              str,
        String[]            allowedUnits)
                                    throws ConfigurationException;

    public ValueWithUnits stringToUnitsWithFloat(
        String              scope,
        String              localName,
        String              typeName,
        String              str,
        String[]            allowedUnits)
                                    throws ConfigurationException;
... continued on the next page
```

Figure 2.15 (continued): The stringTo<Type>() operations

```
... continued from the previous page
    public ValueWithUnits stringToIntWithUnits(
        String              scope,
        String              localName,
        String              typeName,
        String              str,
        String[]            allowedUnits)
                                    throws ConfigurationException;
    public ValueWithUnits stringToUnitsWithInt(
        String              scope,
        String              localName,
        String              typeName,
        String              str,
        String[]            allowedUnits)
                                    throws ConfigurationException;
    ...
}
```

parameter is "Hello, world". If 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 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.

2.9 The List Operations

Figure 2.16 shows the operations for listing the names of entries within a scope. There are two list-type operations: listFullyScopedNames() and listLocallyScopedNames(). However, there are three overloaded versions of each operation, thus making for six variants in total.

The list operations merge the scope and localName parameters to form the fully-scoped name of a scope, and populate the output names

Figure 2.16: The list operations

```
package org.config4j;

public abstract class Configuration
{
    // Type constants
    public static final int CFG_NO_VALUE      = 0;// bit masks
    public static final int CFG_STRING        = 1;// 0001
    public static final int CFG_LIST          = 2;// 0010
    public static final int CFG_SCOPE         = 4;// 0100
    public static final int CFG_VARIABLES     = 3;// STRING|LIST
    public static final int CFG_SCOPE_AND_VARS= 7;// STRING|LIST|SCOPE

    public String[] listFullyScopedNames(
        String       scope,
        String       localName,
        int          typeMask,
        boolean      recursive) throws ConfigurationException;
    public String[] listFullyScopedNames(
        String       scope,
        String       localName,
        int          typeMask,
        boolean      recursive,
        String       filterPattern) throws ConfigurationException;
    public String[] listFullyScopedNames(
        String       scope,
        String       localName,
        int          typeMask,
        boolean      recursive,
        String []    filterPatterns) throws ConfigurationException;
    public String[] listLocallyScopedNames(
        String       scope,
        String       localName,
        int          typeMask,
        boolean      recursive) throws ConfigurationException;
    public String[] listLocallyScopedNames(
        String       scope,
        String       localName,
        int          typeMask,
        boolean      recursive,
        String       filterPattern) throws ConfigurationException;
... continued on the next page
```

Figure 2.16 (continued): The list operations

```
... continued from the previous page
    public String[] listLocallyScopedNames(
        String       scope,
        String       localName,
        int          typeMask,
        boolean      recursive,
        String []    filterPatterns) throws ConfigurationException;
    ...
}
```

parameter with a sorted list of the names of entries in that scope. The boolean `recursive` parameter specifies whether the list operation should recurse into nested sub-scopes. The `typeMask` parameter is a bit mask that specifies which types of entries should be listed. For example, specifying `CFG_VARIABLES` will list only the names of variables, while `CFG_SCOPE` will list only the names of scopes.

By default, a list operation lists the names of *all* the specified entries. However, if one or more *filter patterns* are specified, then the list operation will use the `patternMatch()` operation (Section 2.3 on page 6) to compare each name against the specified patterns, and only names that match at least one filter pattern will be included in the list results.

As an example of the list functions, consider the configuration file shown below:

```
foo {
    timeout = "2 minutes";
    log {
        level = "2";
        file = "/tmp/foo.log";
    };
}
```

The following invocation of `listFullyScopedNames()`:

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

results in `names` containing the following strings:

```
"foo.log"
"foo.log.level"
"foo.log.file"
"foo.timeout"
```

If the same parameters are passed to `listLocallyScopedNames()`, then `names` will contain similar strings, but each string will be missing the `"foo."` prefix.

If you intend to make use of filter patterns, then you should note that filter patterns are matched against the strings that are produced by the list operation. For example, the filter pattern `"time*"` matches against `"timeout"`, which is produced by `listLocallyScopedNames()` in the previous example, but it does *not* match against `"foo.timeout"`, which is produced by `listFullyScopedNames()`. Because of this, you should use `mergeNames()` (Section 2.3 on page 6) to prefix filter patterns with the name of the scope being listed when using `listFullyScopedNames()`. This is illustrated in the following example:

```
String filterPattern = Configuration.mergeNames(scope, "time*");
String names = cfg.listFullyScopedNames(scope, "",
               Configuration.CFG_SCOPE_AND_VARS, true, filterPattern);
```

The list operations call `unexpandUid()`—discussed in Section 2.12 on page 29—on a name before comparing it against filter patterns. Because of this, filter patterns can work with names that have an `"uid-"` prefix. For example, the code below obtains a list of the names of all `uid-recipe` scopes:

```
String filterPattern = Configuration.mergeNames(scope, "uid-recipe");
String names = cfg.listFullyScopedNames(scope, "",
                       Configuration.CFG_SCOPE, true, filterPattern);
```

2.10 Operations for Fallback Configuration

The operations for getting and setting fallback configuration are shown in Figure 2.17.

The one-parameter version of `setFallbackConfiguration()` requires you to create and populate the fallback configuration object yourself. The two- and three-parameter versions creates an initially empty fallback configuration object and then populates it by calling `parse()` with the specified parameters.

2.11 Operations for Security Configuration

As explained in the *Config4* Security* chapter of the *Config4* Getting Started Guide*, Config4* has a built-in security policy that is applied to all `Configuration` objects by default. You can query the current security

Figure 2.17: Operations for Fallback Configuration

```
package org.config4j;

public abstract class Configuration
{
    public void setFallbackConfiguration(Configuration cfg);

    public void setFallbackConfiguration(
        int         sourceType,
        String      source) throws ConfigurationException;

    public void setFallbackConfiguration(
        int         sourceType,
        String      source,
        String      sourceDescription) throws ConfigurationException;

    public Configuration getFallbackConfiguration();
    ...
}
```

policy—which consists of a `Configuration` object and a scope within it—of a `Configuration` object by calling `getSecurityConfiguration()` and `getSecurityConfigurationScope()`, which are shown in Figure 2.18.

You can change the security policy of an individual `Configuration` object by calling `setSecurityConfiguration()`. This operation is overloaded. The first variant shown in Figure 2.18 uses the global scope of an existing `Configuration` object as a security policy.

The second variant enables you to use the specified `scope` of an existing `Configuration` object as a security policy.

The third variant of `setSecurityConfiguration` enables you to specify a file or `"exec#..."` that should be parsed, and combined with the specified `scope` to obtain a security policy.

The details of a security policy are specified by the `allow_patterns`, `deny_patterns` and `trusted_directories` variables in the specified `scope` of the security policy object. See the *Config4* Security* chapter of the *Config4* Getting Started Guide* for details.

Figure 2.18: Operations for Security Configuration

```
package org.config4j;

public abstract class Configuration
{
    public void setSecurityConfiguration(Configuration cfg)
                                        throws ConfigurationException;
    public void setSecurityConfiguration(
        Configuration    cfg,
        String           scope) throws ConfigurationException;
    public void setSecurityConfiguration(String cfgInput)
                                        throws ConfigurationException;
    public void setSecurityConfiguration(
        String           cfgInput,
        String           scope) throws ConfigurationException;

    public Configuration  getSecurityConfiguration();
    public String         getSecurityConfigurationScope();
    ...
}
```

2.12 Operations for the "uid-" Prefix

An identifier that has an "uid-" prefix has both an *expanded* and *unexpanded* form. For example, uid-000000042-recipe is an identifier in its expanded form, while uid-recipe is its unexpanded counterpart. Figure 2.19 lists the operations that Config4J provides for manipulating expanded and unexpanded uid identifiers.

Figure 2.19: Operations for the "uid-" prefix

```
package org.config4j;

public abstract class Configuration
{
    public String expandUid(String str)
                                        throws ConfigurationException;
    public String unexpandUid(String spelling)
                                        throws ConfigurationException;
    public boolean uidEquals(String str1, String str2);
    ...
}
```

Each `Configuration` object keeps an internal counter that starts at 0 and is incremented every time `expandUid()` encounters an `"uid-"` prefix. The current value of that counter is used by `expandUid()` to replace an identifier with its expanded form.

If you populate a `Configuration` object by calling `parse()`, then you are unlikely to need to call `expandUid()`, because the parser invokes that operation automatically whenever it encounters an identifier with an `"uid-"` prefix.

However, if you populate a `Configuration` object by invoking the insertion operations discussed in Section 2.5 on page 12, then it is your responsibility to expand identifiers before invoking the insertion operations.

The `uidEquals()` operation calls `unexpandUid()` for both of its parameters, and tests the resulting names for equality. For example:

```
cfg.uidEquals("uid-000000042-recipe", "uid-recipe")   → true
cfg.uidEquals("uid-000000042-recipe", "uid-employee") → false
```

2.13 The `dump()` Operation

When Config4J parses a configuration file, it stores information about scopes and *name*=*value* pairs in hash tables. Config4J provides a `dump()` operation, shown in Figure 2.20, that converts information in the hash tables into the syntax of a Config4* file.

Figure 2.20: The `dump()` operation

```
package org.config4j;

public abstract class Configuration
{
    public String dump(
         boolean    wantExpandedUidNames,
         String     scope,
         String     localName) throws ConfigurationException;
    public String dump(boolean wantExpandedUidNames);
    ...
}
```

The `dump()` operation is overloaded. The version that takes `scope` and `localName` parameters merges those parameters to form the fully-scoped name of an entry, and then provides a dump of that entry. This

version of `dump()` will throw an exception if the fully-scoped name is of an non-existent entry.

The other version of the operation dumps the entire `Configuration` object.

Both versions of the `dump()` operation take a boolean parameter, `wantExpandedUidNames`, that specifies whether entries that have an `"uid-"` prefix should have their names dumped in expanded or unexpanded form.

Chapter 3

The `SchemaValidator` and `SchemaType` Classes

3.1 The `SchemaValidator` Class

The public and protected operations of the `SchemaValidator` class are shown in Figure 3.1. First, I will discuss the public operations, and then the protected one.

3.1.1 Public Operations

The `getWantDiagnostics()` and `setWantDiagnostics()` operations enable you to get and set a boolean property, the default value of which is `false`. If you set this to `true`, then detailed diagnostic messages will be printed to standard output during calls to `parseSchema()` and `validate()`. These diagnostic messages may be useful when debugging a schema.

The `parseSchema()` operation parses a schema definition and stores it in an efficient internal format. The schema is specified as an array of strings. The `parseSchema()` operation will throw an exception if the parser encounters a problem, such as a syntax error, when parsing the schema.

After you have created a `SchemaValidator` object and used it to parse a schema, you can then call `validate()` to validate (a scope within) a configuration file. If you want, you can call `validate()` repeatedly, perhaps to validate multiple configuration files. The `validate()` operation

Figure 3.1: The SchemaValidator class

```
package org.config4j;

public abstract class Configuration {
    // Type constants
    public static final int CFG_NO_VALUE      = 0;// bit masks
    public static final int CFG_STRING        = 1;// 0001
    public static final int CFG_LIST          = 2;// 0010
    public static final int CFG_SCOPE         = 4;// 0100
    public static final int CFG_VARIABLES     = 3;// STRING|LIST
    public static final int CFG_SCOPE_AND_VARS= 7;// STRING|LIST|SCOPE
    ...
}

public class SchemaValidator
{
    // Values for the forceMode parameter
    public static final int DO_NOT_FORCE   = 0;
    public static final int FORCE_OPTIONAL = 1;
    public static final int FORCE_REQUIRED = 2;

    public SchemaValidator();

    public void    setWantDiagnostics(boolean wantDiagnostics)
    public boolean getWantDiagnostics()

    public void parseSchema(String[] schema)
                                        throws ConfigurationException;
    public void validate(
        Configuration    cfg,
        String           scope,
        String           name,
        boolean          recurseIntoSubscopes,
        int              typeMask,
        int              forceMode) throws ConfigurationException:
    public void validate(
        Configuration    cfg,
        String           scope,
        String           name,
        boolean          recurseIntoSubscopes,
        int              typeMask) throws ConfigurationException;
... continued on the next page
```

Figure 3.1 (continued): The SchemaValidator class

```
... continued from the previous page
    public void validate(
        Configuration    cfg,
        String           scope,
        String           name,
        int              forceMode) throws ConfigurationException;
    public void validate(
        Configuration    cfg,
        String           scope,
        String           name) throws ConfigurationException;
    protected void registerType(SchemaType type)
                                        throws ConfigurationException;
}
```

merges the `scope` and `localName` parameters to form the fully-scoped name of the scope (within the `cfg` object) to be validated.

The `validate()` operation is overloaded several times so that some or all of its final three parameters can be omitted, in which case they are given default values.

The `recurseIntoSubscopes` parameter specifies whether `validate()` should validate only entries in the scope, or recurse down into sub-scopes to validate their entries too. The default value of this parameter is `true`.

The `typeMask` parameter is a bit mask that that specifies which types of entries should be validated. For example, `CFG_VARIABLES` specifies that variables (but not scopes) should be validated. The default value for this parameter is `CFG_SCOPE_AND_VARS`.

By default, `validate()` respects use of the `@optional` and `@required` keywords in the schema. However, if you specify `FORCE_OPTIONAL` for the `forceMode` parameter, then `validate()` will act as if all identifiers in the schema have the `@optional` keyword. Conversely, `FORCE_REQUIRED` makes `validate()` act as if all identifiers without an `"uid-"` prefix in the schema have the `@required` keyword.

3.1.2 Using `registerType()` in a Subclass

Later, in Section 3.2, I will explain how you can implement new schema types. If you implement new schema types, then you will need to write a subclass of `SchemaValidator` to register those new schema types. Figure 3.2 illustrates how to do this.

Registration of new schema types is trivial: the constructor of the

Figure 3.2: A subclass of `SchemaValidator`

```
class SchemaTypeDate { ... }; // Define a new schema type
class SchemaTypeHex { ... };  // Define a new schema type

public class ExtendedSchemaValidator
    extends org.config4j.SchemaValidator
{
    public ExtendedSchemaValidator() {
        super();
        registerType(new SchemaTypeDate());
        registerType(new SchemaTypeHex());
    }
}
```

subclass calls the parent constructor, and then calls `registerType()` to register one instance of each of the new schema types.

Once you have implemented the `ExtendedSchemaValidator` class to register new schema types, your applications need only create an instance of `ExtendedSchemaValidator` (instead of `SchemaValidator`) to be able to make use of those new schema types.

3.2 The `SchemaType` Class

The `SchemaValidator` class perform very little of the validation work itself. Instead, it delegates most of this work to other classes, each of which is a subclass of `SchemaType` (shown in Figure 3.3).

There is a separate subclass of `SchemaType` for each schema type. For example, the Config4J library contains `SchemaTypeBoolean`, which implements the `boolean` schema type, `SchemaTypeInt`, which implements the `int` schema type, and so on.

3.2.1 Constructor and Public Accessors

When the constructor of a subclass of `SchemaType` calls its parent constructor, the parameters specify the name of the schema type and the configuration entry's type, which is one of: `CFG_STRING`, `CFG_LIST` or `CFG_SCOPE`. You can see an example of this in Figure 3.4.

The parameter values passed to the parent constructor are made available via the `getTypeName()` and `getCfgType()` operations shown in Figure 3.3.

Figure 3.3: The SchemaType class

```
package org.config4j;

public abstract class SchemaType implements Comparable
{
    public SchemaType(String typeName, int cfgType);

    public String  getTypeName();
    public int     getCfgType();
    public int     compareTo(Object o); // for interface Comparable

    abstract public void checkRule(
        SchemaValidator    sv,
        Configuration      cfg,
        String             typeName,
        String[]           typeArgs,
        String             rule) throws ConfigurationException;

    public void validate(
        SchemaValidator    sv,
        Configuration      cfg,
        String             scope,
        String             name,
        String             typeName,
        String             origTypeName,
        String[]           typeArgs,
        int                indentLevel) throws ConfigurationException;

    public boolean isA(
        SchemaValidator    sv,
        Configuration      cfg,
        String             value,
        String             typeName,
        String[]           typeArgs,
        int                indentLevel,
        StringBuffer       errSuffix);

    protected SchemaType findType(SchemaValidator sv, String name);
... continued on the next page
```

Figure 3.3 (continued): The SchemaType class

```
... continued from the previous page
    protected final void callValidate(
        SchemaType         target,
        SchemaValidator    sv,
        Configuration      cfg,
        String             scope,
        String             name,
        String             typeName,
        String             origTypeName,
        String[]           typeArgs,
        int                indentLevel) throws ConfigurationException;

    protected final boolean callIsA(
        SchemaType         target,
        SchemaValidator    sv,
        Configuration      cfg,
        String             value,
        String             typeName,
        String[]           typeArgs,
        int                indentLevel,
        StringBuffer       errSuffix) throws ConfigurationException;
}
```

Figure 3.4: Example constructor of a subclass of SchemaType

```
package org.config4j;

class SchemaTypeInt extends SchemaType {
    public SchemaTypeInt() {
        super("int", Configuration.CFG_STRING);
    }
    ...
}
```

The SchemaValidator class invokes registerType() to register an instance of each of the predefined schema types and, as previously shown in Figure 3.2, a subclass of SchemaValidator can invoke registerType() to register instances of additional schema types.

3.2.2 The `checkRule()` Operation

The `SchemaValidator` class invokes the `checkRule()` operation of an object representing a schema type when that type is encountered in a schema rule. I will illustrate this through the schema shown in Figure 3.5.

Figure 3.5: Example schema

```
1  String[] schema = new String[] {
2      "timeout = durationMilliseconds",
3      "fonts = list[string]",
4      "background_colour = enum[grey, white, yellow]",
5      "log = scope",
6      "log.dir = string",
7      "@typedef logLevel = int[0,3]",
8      "log.level = logLevel"
9  };
```

When parsing the first line of the schema, `SchemaValidator` invokes `checkRule()` on the object representing the `durationMilliseconds` schema type. When parsing the next line in the schema, the `SchemaValidator` invokes `checkRule()` on the object representing the `list` schema type, and so on.

Among the parameters passed to `checkRule()` is `typeArgs` (of type `String[]`), which contains the arguments, if any, for the type. This parameter will be an empty array for the rules in lines 2, 5 and 6 of Figure 3.5. For the rule in line 3, `typeArgs` will contain one string (`"string"`); and for the rule in line 4, it will contain three strings (`"grey"`, `"white"` and `"yellow"`). You might think that `typeArgs` should be empty for the rule in line 8. However, the `logLevel` type used in line 8 was defined in line 7 to be `int[0,3]`. Because of this, when `checkRule()` is called for the rule in line 8, `typeArgs` will contain two strings (`"0"` and `"3"`).

The implementation of `checkRule()` must determine whether the strings in `typeArgs` are valid, and throw an exception containing a descriptive error message if not. For example:

- The implementation of `SchemaTypeInt.checkRule()` throws an exception unless: (1) there are *zero* strings in `typeArgs`; or (2) there are *two* strings in `typeArgs`, both strings can be parsed as integers, and the first integer is smaller than or equal to the second integer.
- The implementation of `SchemaTypeList.checkRule()` throws an ex-

ception unless there is exactly one string in `typeArgs`, and that string is the name of a schema type whose configuration entry's type is `CFG_STRING`. This `checkRule()` operation invokes `findType()` to search for the specified schema type; `findType()` returns `null` if the type does not exist.

Deciding whether the `typeArgs` parameter contains acceptable strings is the primary purpose of `checkRule()`. Most of the other parameters are provided to help `checkRule()` make that decision and to format an informative exception message if necessary.

One of the demonstration applications provided with Config4J is called `extended-schema-validator`. That demo contains a class called `SchemaTypeHex` that implements a `hex` (hexadecimal integer) schema type. That class's implementation of `checkRule()` is shown in Figure 3.6. A **bold** font indicates how the operation makes use of parameters.

The only parameter *not* used in the body of the operation is `sv`, which is of type `SchemaValidator`. That parameter is used by the `checkRule()` operation in the `list`, `table` and `tuple` types when invoking `findType()` to determine if items in `typeArgs` are names of types.

3.2.3 The `isA()` and `validate()` Operations

Subclasses of `SchemaType` should implement the `isA()` and `validate()` operations. However, the default implementation of `isA()` is suitable for list-based types, and the default implementation of `validate()` is suitable for string-based types. Because of this, a subclass of `SchemaType` needs to implement only one of these two operations.

3.2.3.1 String-based Types: `isA()`

If you are providing schema support for a string-based type, then you must implement the `isA()` operation. Among the parameters passed to this operation is a `string` called `value`; the `isA()` operation should return `true` if `value` can be parsed as the schema type. For example, the `SchemaTypeInt::isA()` operation returns `true` for `"42"` and returns `false` for `"hello, world"`.

If `isA()` returns `false`, then the operation can optionally set the `errSuffix` parameter (which is of type `StringBuffer`) to be a descriptive message that explains *why* the string is not suitable. This message will be appended to an exception message.

Figure 3.6: Implementation of `SchemaTypeHex.checkRule()`

```
public class SchemaTypeHex extends SchemaType
{
    public void checkRule(
        SchemaValidator    sv,
        Configuration      cfg,
        String             typeName,
        String[]           typeArgs,
        String             rule) throws ConfigurationException
    {
        int                len;
        int                maxDigits;

        len = typeArgs.length;
        if (len == 0) {
            return;
        } else if (len > 1) {
            throw new ConfigurationException("schema error: the '"
                        + typeName + "' type should take either no "
                        + "arguments or 1 argument (denoting "
                        + "max-digits) in rule '" + rule + "'");
        }
        try {
            maxDigits = cfg.stringToInt("", "", typeArgs[0]);
        } catch(ConfigurationException ex) {
            throw new ConfigurationException("schema error: "
                    + "non-integer value for the 'max-digits' "
                    + "argument in rule '" + rule + "'");
        }
        if (maxDigits < 1) {
            throw new ConfigurationException("schema error: the "
                    + "max-digits argument must be 1 or greater in "
                    + "rule '" + rule + "'");
        }
    }
    ...
}
```

Figure 3.7 illustrates how `isA()` might be implemented for a schema type that denotes hexadecimal integers. A **bold** font indicates how the operation makes use of parameters. This implementation of `isA()` contains two straightforward checks. First, it checks whether `value` consists

of hexadecimal digits. Second, if `typeArgs` specifies a maximum number of digits, then `isA()` checks if this limit has been exceeded.

Figure 3.7: Implementation of `isA()` for a `hex` type

```
public class SchemaTypeHex extends SchemaType
{
    public boolean isA(
        SchemaValidator    sv,
        Configuration      cfg,
        String             value,
        String             typeName,
        String[]           typeArgs,
        int                indentLevel,
        StringBuffer       errSuffix) throws ConfigurationException
    {
        int                    maxDigits;

        if (!isHex(value)) {
            errSuffix.append("the value is not a hexadecimal number");
            return false;
        }
        if (typeArgs.length == 1) {
            //--------
            // Check if there are too many hex digits in the value
            //--------
            maxDigits = cfg.stringToInt("", "", typeArgs[0]);
            if (value.length() > maxDigits) {
                errSuffix.append("the value must not contain more "
                                 + "than " + maxDigits + " digits");
                return false;
            }
        }
        return true;
    }

    public static boolean isHex(String str) {
        ... // implementation will be shown later in this chapter
    }
    ...
}
```

3.2.3.2 List-based Types: `validate()`

Config4* has three built-in, list-based schema types: `list`, `tuple` and `table`. Each of these schema types takes arguments, for example:

```
String[] schema = new String[] {
    "@typedef money = units_with_float[\"£\", \"$\", \"€\"]",
    "fonts      = list[string]",
    "point      = tuple[float,x, float,y]",
    "price_list = table[string,product, money,price]"
};
```

Each of those list-based schema types implements `validate()` in a similar way, so I will discuss only the implementation for the `table` schema type, using the definition of `price_list` in the above example.

- A call of `cfg.lookupList(scope, name)` is made to retrieve the value of the list variable from the configuration object.

- The `typeArgs` parameter contains all the arguments to the schema type (`"string"`, `"product"`, `"money"` and `"price"` for the `price_list` variable in the example). Those pairs of strings define the types and names of columns within the table. The `validate()` operation checks that the length of the list is a multiple of the number of columns in the table's definition.

- Finally, `validate()` iterates over all the items in the list. For each item, `validate()` calls `findType()` for the item's column type (obtained from `typeArgs`) to retrieve the item's schema type; it invokes the `isA()` operation of that type, and throws an exception if `isA()` returns `false`.

  The invocation of `isA()` is not made directly. Rather, it is made indirectly by invoking `callIsA()`, which is shown in Figure 3.3 on page 37. Doing this ensures that diagnostic messages can be printed if the `SchemaValidator` was created with `true` specified for the `wantDiagnostics` constructor parameter.

If you want to implement schema support for a list-based type, then you should implement the `validate()` operation in a manner similar to that described above. I recommend that you examine the source code of the `SchemaTypeList`, `SchemaTypeTable` or `SchemaTypeTuple` class for concrete details.

3.3 Adding Utility Operations to a Schema Type

The infrastructure within Config4J to support a built-in data type is split over three classes:

- The `SchemaType<Type>` class implements the schema validation infrastructure.
- The `SchemaValidator` class calls `registerType()` to register each schema type.
- The `Configuration` class provides operations with names of the form `lookup<Type>()`, `is<Type>()` and `stringTo<Type>()`.

In this chapter, I have explained how you can provide schema validation support for a new type by writing a `SchemaType<Type>` class and registering it in a subclass of `SchemaValidator`. However, I have not yet explained how you can write a subclass of `Configuration` to implement the `lookup<Type>()`, `is<Type>()` and `stringTo<Type>()` operations.

The `Configuration` class is an abstract base class, and its static `create()` operation creates an instance of a hidden, concrete subclass. This enforces a separation between the public API and the implementation details of Config4*. Most of the time, this separation is beneficial. However, it has a drawback: you cannot write a subclass of `Configuration` to add additional operations, such as `lookup<Type>()`, `is<Type>()` and `stringTo<Type>()`.

A good way to workaround this drawback is to define the desired functionality as `static` operations in the `SchemaType<Type>` class. For example, if you are writing a class called `SchemaTypeHex` (for hexadecimal integers), then you can implement `lookupHex()`, `isHex()`, and `stringToHex()` as `static` operations in the `SchemaTypeHex` class. This is illustrated in Figure 3.8.

Figure 3.8: Utility operations in the SchemaTypeHex class

```
public class SchemaTypeHex extends SchemaType {
    public SchemaTypeHex() {
        super("hex", Configuration.CFG_STRING);
    }
    public static int lookupHex(
        Configuration    cfg,
        String           scope,
        String           localName) throws ConfigurationException
    {
        String str = cfg.lookupString(scope, localName);
        return stringToHex(cfg, scope, localName, str);
    }
    public static int lookupHex(
        Configuration    cfg,
        String           scope,
        String           localName,
        int              defaultVal) throws ConfigurationException
    {
        if (cfg.type(scope, localName) == Configuration.CFG_NO_VALUE) {
            return defaultVal;
        }
        String str = cfg.lookupString(scope, localName);
        return stringToHex(cfg, scope, localName, str);
    }
    public static int stringToHex(
        Configuration    cfg,
        String           scope,
        String           localName,
        String           str,
        String           typeName) throws ConfigurationException
    {
        try {
            return (int)Long.parseLong(str, 16);
        } catch(NumberFormatException ex) {
            throw new ConfigurationException(cfg.fileName()
                        + ": bad " + typeName + " value ('" + str
                        + "') specified for '"
                        + cfg.mergeNames(scope, localName) + "'");
        }
    }
... continued on the next page
```

Figure 3.8 (continued): Utility operations in the SchemaTypeHex class

```
... continued from the previous page
    public static int stringToHex(
        Configuration    cfg,
        String           scope,
        String           localName,
        String           str) throws ConfigurationException
    {
        return stringToHex(cfg, scope, localName, str, "hex");
    }

    public static boolean isHex(String str)
    {
        try {
            Long.parseLong(str, 16);
            return true;
        } catch(NumberFormatException ex) {
            return false;
        }
    }
    ... // checkRule() and isA() were shown earlier in this chapter
}
```