Overview of Java application configuration frameworks

Victor S. Denisov

Abstract **— This paper reviews three major application configuration frameworks for Java-based applications: java.util.Properties, Apache Commons Configuration and Preferences API. Basic functionality of each framework is illustrated with code examples. Pros and cons of each framework are described in moderate detail. Suggestions are made about typical use cases for each framework.**

***Keywords* — application configuration management, application settings management, framework, Java.**

I. Introduction

Most software applications being developed today are designed to run in several different environments; environments can differ both by the stage in the project's lifecycle (i.e., developer's workstation, CI server, testing/QA deployment, staging, production) and by the actual production environment (from different operating systems to different user requirements). Application Configuration Management (also known as Application Settings Management) assumes that application's behavior can be influenced by a number of configuration attributes, which, when referenced by the application code, allow it to match specifics of its environment.

Application Configuration Management can be roughly divided into four primary functional areas:

- configuration attributes storage and retrieval (both at runtime and between application executions);
- reference of configuration attributes from inside the application;
- attribute modification by end-users (either from inside the application itself or via a standalone/dedicated application);
- report on current configuration, including defined attributes, their values, etc.

Traditionally, the first two areas – which are pre-requisites for the last two – generated the most interest from developers, resulting in emergence of several general-purpose configuration frameworks. Lacking an accepted ACM standard, those frameworks differ significantly in almost all conceivable aspects – from development platform to implementation complexity to robustness to available attribute schemes to supported storage facilities. However, they all share a common design goal – to make application configuration management easier for both developers and end-users.

Manuscript received June 13, 2013.

Victor S. Denisov is with Lomonosov Moscow State University, Department of Computational Mathematics and Cybernetics, Moscow, Russia (e-mail: vdenisov@plukh.org)

II. What's included in the overview (and what's not).

This paper provides overview of the three most commonly used frameworks for managing configuration information for Java-based applications: java.util.Properties [1], Apache Commons Configuration [2] and Preferences API [3][4]. Those are, by far, the most widely used configuration frameworks in the Java world – or, at least, in the open-source part of it, judging by the simple count of top Apache, GitHub and SourceForge projects using each particular framework.

Of course, there are more configuration frameworks for Java applications than can be listed in a limited space of this paper. Two of the more well-known frameworks are jFig [5] and jConfig [6] (which, sadly, seems to be abandoned at the moment).

Finally, it is worth mentioning that there are a number of wrappers around the mentioned frameworks which provide various additional benefits for the developer, such as type safety, annotations-driven configuration definition, etc (see, for example, [7] and [8]). While not on a level of a fully-matured framework (both in terms of functionality, complexity and adoption ratio), they do provide some attractive (even if minor) features, making their use warranted in certain specific cases.

III. The father of them all - java.util.Properties

Properties class was the first configuration "framework" for Java, being present in Java API from the very first 1.0 release back in 1996. It established a number of important conventions for storing and retrieving configuration information:

- configuration attributes were implemented as a flat collection of name-value pairs (collectively called properties); only strings could be used for both names and values;
- attributes could be loaded from and saved to any Java InputStream/OutputStream implementation – most often, FileInputStream/FileOutputStream for storing properties in a file on a local file system;
- properties file format (with a default .properties

extension) was simple and well documented, which made possible alternative implementations and implementations in other programming languages;

- a concept of a default value for a property was introduced – a value returned if this property wasn’t loaded from properties file or otherwise explicitly defined by the application;
- a pseudo-hierarchical dot-separated property naming convention was recommended (but not enforced) by Properties documentation and accompanying examples.

The best thing about java.util.Properties is that it’s trivially easy to use. Consider this example (for full source code of examples in this article, see https://github.com/vdenisov/config-overview-examples, this is from Example1.java in "example-properties" project):

```
//Instantiate properties object and set some
property values
final Properties properties = new Properties();
properties.setProperty("property", "value");
properties.setProperty("another.property",
"value2");

//Store properties to file
try (FileOutputStream out = new
FileOutputStream("example1.properties")) {
    properties.store(out, "Example1 properties");
} catch (IOException e) {
    e.printStackTrace();
}
```

The above example assigns values to properties named "property" and "another.property" (Properties class doesn’t differentiate between attribute definition and value assignment), then stores the configuration in a file named "example1.properties":

```
#Example1 properties
#Thu Jun 13 21:19:29 MSK 2013
another.property=value2
property=value
```

This is a plain text file, which can be edited by any common text editor, as well as passed around from system to system by whatever method is deemed to be more convenient (from plain old removable disks to remote cloud storage). Note that the order in which properties will be enumerated – including enumeration when saving – is not guaranteed.

Reading properties is equally simple (see Example2.java):

```
//Instantiate properties object
final Properties properties = new Properties();

//Load properties from file
try (FileInputStream in = new
FileInputStream("example1.properties")) {
    properties.load(in);
} catch (IOException e) {
    e.printStackTrace();
}
```

Unfortunately, that’s about it when it comes to java.util.Properties features. On the other hand, this class suffers from a number of design and implementation features, two of the most important ones being no validation and type-safety whatsoever and confusing public interface.

Let's deal with the second issue first. Properties class extends java.util.Hashtable and inherits all of its public methods. However, new methods introduced by java.util.Properties have some unexpected semantics when used together with Hashtable's methods (see Example3.java):

```
//Instantiate properties object and set some
property values
final Properties properties = new Properties();
properties.put("property", 1); //Will produce an
unexpected result later

//Output property values (somewhat unexpected
result)
System.out.println("property=" +
properties.getProperty("property"));
```

In the above example, result of properties.getProperty(…) call will return null – which is somewhat counterintuitive. First, we put integer value of 1 into the hashtable which backs the properties map; next, we invoke getProperty(…) method and expect it to either return a String value of "1" (or whatever toString() method call for the appropriate object instance would produce) or an exception (since Integer is not assignment-compatible with String). Quite unexpectedly, Properties class treats all hashtable entries with non-String values as being absent from the collection altogether, thus returning null for the above call.

Another problem is that Properties puts all responsibility to validate and convert property values to and from their String representations on application developer. Since it also doesn't support any sort of self-documentation (neither run-time, such as annotations, nor compile-time, such as generics), this leads to extremely non-obvious errors, such as assuming different valid value ranges for a certain property, or assuming similar, but slightly different, encoding schemes for binary property values in different parts of the code (see Example4.java).

Finally, java.util.Properties has no built-in method for propagating configuration changes inside the application – again, responsibility to notify different application components about changes in configuration lies with the developer.

To sum it up, the pros and cons of java.util.Properties as configuration framework are as follows:

Pros:

- takes literally one line of code to instantiate and use;
- provides simple key-value mapping;
- well-defined storage standard;
- easy to change stored configuration information with external tools.

Cons:

- no type safety;
- not self-documenting;
- responsibility for validation and value conversion lies

with the application;

- no way to monitor configuration file for changes;
- no configuration change listeners.

All in all, java.util.Properties can only be recommended for small (less that approximately 20 classes) projects. Projects which start big, or which will probably grow over time, should look to another way of handling their configuration requirements.

IV. Apache Commons Configuration

Apache Commons Configuration (CC for short) started its life as a set of configuration classes for Apache JServ. It then served a number of Apache Foundation projects, until finally becoming part of the Apache Commons library collection in 2003.

Perhaps the most important difference with java.util.Properties is that CC uses a factory pattern to hide a variety of different configuration implementations behind a single Configuration interface, thus isolating configuration-specific logic behind a common facade.

CC provides a number of classes implementing Configuration interface, which support various persistent storage formats and mediums (such as properties files, XML documents, JNDI, JDBC), as well as extend basic Configuration interface contract in several ways (such as adding support for hierarchical information, or merging configuration information from several other Configuration implementations).

Despite all that additional functionality, CC is still trivially easy to use. It takes just a couple lines of code to instantiate a Configuration implementation, define a property and persist the resulting set to the local filesystem (see Example1.java in " example-commons-config" project):

```
final PropertiesConfiguration configuration = new
PropertiesConfiguration();
configuration.setProperty("property", "value");
configuration.setHeader(
    "Example1 properties configuration");
try {
    configuration.save("example1.properties");
} catch (ConfigurationException e) {
    e.printStackTrace();
}
```

Loading configuration information from a file is equally easy. Other persistence methods may require a little more setup, but, for the most part, they're just as straightforward as file-based storage methods (see Example2.java):

```
final PropertiesConfiguration conf = new
PropertiesConfiguration();
try {
    conf.load("example1.properties");
} catch (ConfigurationException e) {
    e.printStackTrace();
}
System.out.println("property=" +
conf.getString("property"));
```

Another interesting feature of Commons Configuration is its ability to maintain and control the layout of configuration files. Each PropertiesConfiguration object is associated with Layout, an instance of the PropertiesConfigurationLayout class. This class is responsible for preserving, to the extent possible, the original structure of the file, including property ordering, comments, extra lines, etc. When configuration is persisted back to the filesystem, Layout ensures that all specific layout restrictions (such as a separator char between property names and values) are enforced (see Example3.java for an example of persisting configuration with and without changes to the layout).

Commons Configuration has built-in support for working with lists (or arrays) of objects as values of a single property (see Example4.java):

```
final PropertiesConfiguration conf = new
PropertiesConfiguration();
conf.setProperty("array_property",
    "red, green, blue");
String[] strings =
conf.getStringArray("array_property");
System.out.println("Got " + strings.length +
" elements, first is " + strings[0] +
", second is " + strings[1] + " and third is " +
strings[2]);
```

As can be seen from the example above, CC automatically splits string values around commas – which is the default delimiter character for multi-valued properties. Same thing happens when properties are read from file; when saving, values of multi-valued properties are concatenated using delimiter chars, forming a single property entry.

Finally, CC supports configuration listeners, which receive configuration change events whenever configuration information maintained by the object they're attached to changes (for example, when a new property is added or an existing property value is updated). See Example5.java for an example of how configuration listener can be setup and how configuration change events can be processed by the application.

Here are the pros and cons of Commons Configuration framework:

Pros:

- easy to use;
- provides robust structured key-value mapping;
- excellent capabilities when working with file-based configuration sources;
- allows storing and retrieving configuration information to/from a variety of local and remote sources;
- limited type-safety for most common value types;
- notifications on configuration information changes;
- easy to change stored configuration information with external tools.

Cons:

- additional dependencies make the framework impractical for small projects;
- not self-documenting;

- very limited capabilities for value validation;

Basically, for all but the smallest Java projects there is no reason to prefer java.util.Properties over Commons Configuration – it provides significantly more features while keeping the same ease of use that Properties are known for.

V. Preferences API

Preferences API (along with its main implementing class, java.util.prefs.Preferences) first appeared in Java2 SE 1.4, released in 2002. Its main goal was to provide platform-independent API for storing and retrieving application configuration to/from a platform-appropriate backing store (like a registry service on Windows, XML files on *nix, etc).

Preferences API is similar to both Properties and Commons Configuration in that it allows the application to store its configuration information as a set of key-value pairs. Like CC, Preferences supports structured information, with inner nodes identifying the application and/or specific application classes to which specified information sets belong.

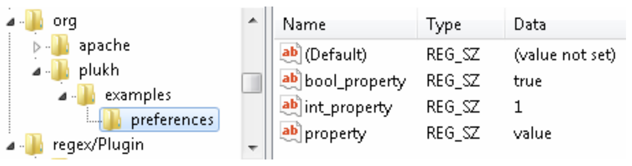
Unlike other configuration frameworks, Preferences makes clear distinction between "system" and "user" backing stores. System store must be accessible to all users of the system; user stores are unique for each distinct user of the underlying OS. This allows to separate global configuration elements (such as the path to and version of the application) and user-specific elements (such as window position and dimension, document history, font preferences, etc).

Using Preferences API is relatively simple. Usually, the application obtains Preferences object for its main class (or otherwise identifies the parent node storing its configuration information), then stores and retrieves key-value pairs for this node and any subnodes it requires. In Example1.java in "example-preferences" project a Preferences instance for the user preferences of the main class (residing in the package "org.plukh.examples.preferences") is obtained and a couple of configuration properties are added to it:

```
Preferences prefs =
Preferences.userNodeForPackage(Example1.class);

prefs.put("property", "value");
prefs.putInt("int_property", 1);
prefs.putBoolean("bool_property", true);
```

On Windows machine, it results in several keys being created and/or updated in the registry (see Fig. 1):



```
org
  apache
  plukh
    examples
      preferences
regex/Plugin
```

Name	Type	Data
(Default)	REG_SZ	(value not set)
bool_property	REG_SZ	true
int_property	REG_SZ	1
property	REG_SZ	value

Fig. 1

When executed on a Unix machine, this same code would result in a series of hidden directories created in the user's home directory, the last of which would contain an XML document with actual configuration properties.

Note that, unlike in previously described frameworks, there are no explicit calls to persist configuration information to the backing store. If you'll check Example2.java, you'll notice that there are no explicit calls to load the information before accessing it as well. Preferences API implementation must manage persistence for the application, loading and saving information to the backing store as needed.

Unfortunately, specifics of the backing store implementation on different platforms aren't standardized. This poses two critical problems. First, it's difficult to access configuration information managed by Preferences API consistently across different platforms with external tools (for example, to allow for centralized management of configuration information) – different set of tools (probably using different technologies) should be developed for all supported platforms. Second, there is no guarantee that different Preferences implementations (from different JVM vendors, and even different versions provided by the single vendor) will be interoperable – and so there is no guarantee that configuration information will not be corrupted or lost in a routine JVM upgrade, for example.

Preferences API allows to specify a custom Preferences class implementation (or, rather, a factory for creating Preferences instances via the PreferencesFactory class). This is usually used to create alternative backing store implementations (see [9] for an implementation which uses plain .properties files to store configuration information); another reason is to provide read-only stores (so that applications can retrieve, but not modify, their configuration). However, only a single PreferencesFactory instance can be configured for a JVM instance, which limits the applicability of this approach for virtual machines with several independent applications running at once (such as J2EE application servers – for an old, but still relevant, discussion, see [10]).

Preferences API provides limited type safety by safely converting string property values read from the backing store to the most common Java types, such as boolean or int – using default values (which are mandatory in all "get"-type calls) if conversion fails. However, it doesn't enforce specific value types for specific properties, so the code below is perfectly valid, even if obviously error-prone (see Example4.java):

```
prefs.putInt("property", 1);
prefs.put("property", "string");
```

Additionally, requirement to specify default values on every call (rather than when application is first initialized, or when the configuration is first persisted to the backing store) leads to error-prone code, where different default values can be used in different calls – which, depending on context, can be both correct and incorrect. Consider Example5.java. In inner class Right, different defaults are used based on whether or not the app runs on Windows OS, which is probably what the developer intended. In inner class Wrong, there is a typo, and different defaults are erroneously specified in two different calls – in larger applications, successfully isolating such problems can require significant effort.

Summing it up, pros of the Preferences API:

- built into JRE – no external dependencies;
- robust structured key-value mapping;
- backing store implementation is fully isolated from the application;
- limited type-safety for most common value types;
- notifications on configuration information changes;
- thread-safe serialized access to configuration information inside a single JVM; guarantee against backing store corruption with multi-JVM access.

Cons:

- requirement to specify a default value in all "get" calls makes for error-prone code if the same property is referenced from multiple parts of the code;
- platform-specific, proprietary and not always predictable backing store format makes it difficult to use external tools to manage configuration information in a platform-independent way;
- implementation-specific backing store - configuration can be lost on JVM vendor/version change;
- can't use different configuration sources simultaneously without lots of additional code;
- no isolation between information for different applications, both inside the same VM and across VMs.

Preferences API is clearly targeted at desktop applications running on a limited number of platforms, with each application running in a single dedicated VM. It's easy to use and has some extremely useful features, like automatic persistence management. It also suffers from a number of issues which make it significantly less suited for server-side or embedded applications.

VI. Conclusion

There are three primary application configuration frameworks in use now in the Java ecosystem: java.util.Properties, Apache Commons Configuration and the Preferences API. While java.util.Properties is severely outdated (but is still widely used), both Commons Configurations and Preferences API provide a far more robust and feature-rich – but still easy to use – alternative.

Unfortunately, both Commons Configuration and Preferences API still suffer from several common problems:

- not self-documenting – information about each property's type, acceptable values, usage semantics (such as being read-only), etc has to be documented manually (in comments or in external documentation); the code which references those properties is too generic;
- limited type-safety – while both frameworks provide safe value conversions for common value types, such as numbers or booleans, they don't enforce type-safety on assignment;
- limited extensibility – no built-in mechanisms for extending the framework with support for new data types, data structures, etc;
- no support for complex data structures, such as JavaBeans, as property values.

VII. Future Work

Shortcomings of existing Java ACM frameworks call for a more up-to-date application configuration framework – something which will be discussed in-depth in future papers, starting with the discussion of requirements and followed by an actual prototype configuration framework implementation.

References

[1] Class java.util.Properties [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html

[2] Commons Configuration [Online]. Available: http://commons.apache.org/proper/commons-configuration/

[3] Preferences API Overview [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/preferences/overview.html

[4] Class java.util.prefs.Preferences [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/util/prefs/Preferences.html

[5] jFig – Java Configuration Solution [Online]. Available: http://jfig.sourceforge.net/

[6] jConfig [Online]. Available: http://www.jconfig.org/

[7] OWNER - Java™ properties files made super simple! [Online]. Available: http://lviggiano.github.io/owner/

[8] Hughes, M. (2007, Jul 24). Easing configuration. IBM developerWorks Technical Library [Online]. Available: http://www.ibm.com/developerworks/java/library/j-configint/index.html

[9] Croft, D. (2009, Jun 18). Java Preferences using a file as the backing store [Online]. Available: http://www.davidc.net/programming/java/java-preferences-using-file-backing-store

[10] Halloway, S. D. (2002, Aug 30). Java Properties Purgatory Part 2 [Online]. Available: http://www.informit.com/articles/article.aspx?p=29011