
Perforce 2009.1
P4Java API User's Guide

August 2009

This manual copyright 2009 Perforce Software.

All rights reserved.

Perforce software and documentation is available from <http://www.perforce.com>. You may download and use Perforce programs, but you may not sell or redistribute them. You may download, print, copy, edit, and redistribute the documentation, but you may not sell it, or sell any documentation derived from it. You may not modify or attempt to reverse engineer the programs.

Perforce programs and documents are available from our Web site as is. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by Perforce Software.

Perforce Software assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

By downloading and using our programs and documents you agree to these terms.

Perforce and Inter-File Branching are trademarks of Perforce Software. Perforce software includes software developed by the University of California, Berkeley and its contributors.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

Table of Contents

Chapter 1	P4Java Programming.....	5
	About P4Java.....	5
	System Requirements.....	5
	Installation.....	5
	Documentation.....	6
	Java package roadmap.....	6
	Basic P4Java usage model.....	7
	Typical usage patterns.....	9
	The server factory, P4JServer, and P4JClient interfaces.....	9
	Perforce file operations.....	10
	Advanced usage notes.....	12
	Perforce server addresses, URIs, and properties.....	12
	The P4JServerResource Interface.....	12
	P4Java properties.....	13
	Character Set Support.....	14
	Error Message Localization.....	15
	Logging and tracing.....	15
	Exceptions and errors.....	15
	Standard implementation classes.....	15
	I/O and file metadata issues.....	15
	Threading issues.....	16
	Authentication.....	17
	Other Notes.....	17

About P4Java

Perforce Software's P4Java is a Java API that enables applications to access Perforce's software configuration management (SCM) system and features in a "Java natural" and Java-native way. P4Java presents Perforce services and Perforce-managed resources and files as first-class Java interfaces, classes, methods, and objects, rather than as simple strings or command-line-style functions. This approach makes it easier to integrate the API into Java applications and tools, and is particularly useful for integrating Perforce into model-view-controller (MVC) contexts and workflows.

P4Java is aimed mostly at the following types of Java development:

- Standalone applications that need to access Perforce services from within the application
- Plug-ins for Java tools such as Eclipse, ant, Cruise Control, and so on, that need to communicate with Perforce servers
- J2EE applications, where P4Java can be embedded within a servlet and/or presented as a web service or an AJAX binding for client-side use

This document provides a brief guide to installing and using P4Java, and assumes a basic knowledge of both Java (JDK 5 or later) and Perforce SCM features.

System Requirements

P4Java assumes the presence of a JDK 6 or later environment, but will work against a JDK 5 installation, with some limitations. For details, refer to the P4Java release notes.

Installation

Download the P4Java zip file from the Perforce web site, extract the enclosed JARs and other files to a temporary directory, then install the `p4java.jar` JAR file into a location that is suitable for use by compilers, JVMs, and other development tools or applications.

Documentation

Included with the P4Java zip file is a directory of documentation that contains this document and a full Javadoc document set for all public interfaces and classes.

The Javadoc document set can be found at:

www.perforce.com/perforce/doc.091/manuals/p4java-javadoc/index.html

Java package roadmap

The P4Java API contains the following main public packages:

- `com.perforce.p4java`: the main P4Java package hierarchy root. Contains a small handful of API-wide definitions and classes for activities like logging, tracing, and package metadata.
- `com.perforce.p4java.server`: contains the server factory class and Perforce P4JServer server interface, and associated classes and interfaces related to the P4JServer definition. This package enables participating applications to connect to Perforce servers and start interacting with Perforce services through the P4JServer interface.
- `com.perforce.p4java.client`: defines the Perforce P4JClient client interface and associated classes and support definitions. Participating applications typically use the P4JClient interface to access Perforce client services such as syncing and adding, editing, or deleting files.
- `com.perforce.p4java.exception`: defines the main publicly-visible exceptions likely to be encountered in general use, and some specialized and rarely-encountered errors.
- `com.perforce.p4java.core`: contains interface definitions for major Perforce-managed objects such as changelists, jobs, and clients.
- `com.perforce.p4java.core.file`: contains the main Perforce P4JFileSpec interface for accessing and defining the various types of files that Perforce manages (for example, `depot`, `local`, and `client`), along with associated definitions.
- `com.perforce.p4java.impl.generic`: root package for “generic” or standard implementations of many useful Perforce client, changelist, job, and similar interfaces. These implementations are available for use by participating applications, but are not mandatory.

Basic P4Java usage model

The following basic model for P4Java reflects typical Perforce SCM usage:

1. A Java application uses the P4Java `P4JServerFactory` class to obtain a `P4JServer` interface onto a specific Perforce server at a known network address and port, and connects to this Perforce server through the `P4JServer` interface that is returned from the factory.
2. The application optionally logs in to the Perforce server through the `P4JServer`'s login and associated methods.
3. The application obtains a suitable `P4JClient` interface into a Perforce client workspace through the `P4JServer` interface's "get client" method.
4. The application syncs the Perforce client workspace through the `P4JClient` interface's sync method.
5. The application gets and processes (Java `java.util.List`) lists of depot, client, and local files in (or associated with) the Perforce client workspace, through the `P4JServer` and `P4JClient` interfaces.
6. The application adds, edits, or deletes files in the local Perforce client workspace using the `P4JClient` interface. These files are added to the default or a numbered Perforce changelist represented by one or more `P4JChangeList` interfaces, which are obtained through the `P4JClient` or `P4JServer` interfaces. (There are often several ways to obtain a specific type of object depending on context, but these tend to be convenience methods rather than fundamental.)
7. The application submits a specific changelist using the associated `P4JChangeList` interface. This submission can be linked with one or more Perforce jobs, represented by the `P4JJob` interface.
8. The application can switch between Perforce workspaces, browse Perforce jobs and changelists, log in as a different user, and add, edit, or delete files, using the relevant `P4JServer` or `P4JClient` interfaces.
9. To disconnect from a Perforce server, the application calls the `disconnect` method on the `P4JServer` interface.

This usage model relies heavily on representing all significant Perforce objects – clients, servers, changelists, jobs, files, revisions, labels, branches, and so on – as first-class Java interfaces, classes, or enums, and, where appropriate, returning these objects as ordered Java lists so that the developer can iterate across the results using typical Java iterator patterns. P4Java uses JDK 5 (and later) parameterized types for these lists.

P4Java represents most recoverable usage errors and Perforce errors as Java exceptions that are subclassed out of the main `P4JException` class, and thrown from nearly every significant `P4JServer` and `P4JClient` interface method (and from subsidiary and associated class methods). Most such errors are connection errors (caused by a network or connectivity issue), access errors (caused by permissions or authentication issues), or request errors (caused by the Perforce server detecting a badly-constructed request or non-existent file spec). P4Java applications catch and recover from these errors in standard ways, as discussed in “Exceptions and errors” on page 15.

Exceptions are not used in methods that return multiple files in lists, because the server typically interpolates errors, informational messages, and valid file specs in the same returns. P4Java provides a single method call as a standard way of identifying individual returns in the (often very long) list of returns, discussed in detail in “Perforce file operations” on page 10.

In general, the methods and options available on the various P4Java API interfaces map to the basic Perforce server commands (or the familiar `p4` command line equivalent), but there are exceptions. Not all Perforce server commands are available through the P4Java API.

Unlike the Perforce C++ API or the `p4` command-line client, P4Java is not intended for direct end-user interaction. Rather, P4Java is intended to be embedded in GUI or command-line applications to provide Perforce client / server communications, and P4Java assumes that the surrounding context supplies the results of user interaction directly to P4Java methods as Java objects. Consequently, many of the environment variables used by command-line client users (such as `P4PORT` or `P4USER`) are deliberately ignored by P4Java. The values they usually represent must be explicitly set by appropriate `P4Jserver` methods or other methods.

The standard default P4Java server and client implementations are basically thread-safe. To avoid deadlock and blocking, refer to “Threading issues” on page 16.

Typical usage patterns

This section briefly describes typical usage patterns and provides a starting point for developers using P4Java for the first time.

The server factory, P4JServer, and P4JClient interfaces

The `com.perforce.p4java.server.P4JServer` interface represents a specific Perforce server in the P4Java API, with methods to access typical Perforce server services. Each instance of a `P4JServer` interface is associated with a Perforce server running at a specified location (network address and port), and each `P4JServer` instance is obtained from the P4Java server factory, `com.perforce.p4java.server.P4JServerFactory`, by passing it a suitable server URI and optional Java properties:

```
try {
    P4JServer server = P4JServerFactory.getServer(
        "p4java://localhost:1666", null);
    server.connect();
    server.setUserName("demouser");
    server.login("demopassword");

    // display client address details...
    P4JServerInfo serverInfo = server.getServerInfo();
    System.out.println(serverInfo.getClientAddress());

    // List known depots:
    List<P4JDepot.P4JDepot> depotList = server.getDepotList();
    for (P4JDepot.P4JDepot depot : depotList) {
        // Process Perforce depot metadata...
    }
    // Process user commands ...
    for (String cmd : userCmds) {
        // do command...
    }
    server.logout();
    server.disconnect();
} catch (P4JConfigException p4jce) {
    // Process a client-side configuration error...
} catch (P4JConnectionException p4jce) {
    // Process the inability to connect (e.g. network or URI issue)
} catch (P4JResourceException p4jce) {
    // Process the lack of factory resources...
} catch (P4JNoSuchObjectException p4jce) {
    // Process the non-existence of the requested implementation...
} catch (P4JAccessException p4jce) {
    // Process login failure or other permissions issue...
} catch (P4JRequestException p4jce) {
    // Process Perforce command request error...
}
}
```

Multiple `P4JServer` objects can represent the same physical Perforce server, and this approach is recommended for heavyweight usage and for multi-threaded applications.

The Java properties parameter passed to the factory in the first example is null, but you can pass in a variety of generic and implementation-specific values as described in “Character Set Support” on page 14.

Perforce client workspaces are represented by the `com.perforce.p4java.client.P4JClient` interface, which can be used to issue Perforce client workspace-related commands such as sync commands, file add /delete / edit commands, and so on. A `P4JClient` interface is typically obtained from a `P4JServer` interface using the `getClient()` method, and is associated with the `P4JServer` using the `setCurrentClient()` method:

```
try {
    P4JClient client = server.getClient("democlient");
    if (client != null) {
        server.setCurrentClient(client);
    }
} catch (P4JException exc) {
    // Process errors...
}
```

Unlike the `p4` command line client, there are no defaults for user and workspace. Your application must explicitly associate a workspace (a `P4JClient` client object) and user with the server.

Perforce file operations

To define common Perforce-managed file attributes and options, P4Java uses the `com.perforce.p4java.core.file.P4JFileSpec` interface. Attributes like revisions, dates, actions, and so on, are also defined in the `core.file` package, along with some key helper classes and methods. In general, most Perforce file-related methods are available on the `P4JServer` and `P4JClient` interfaces, and might also be available on other interfaces such as the `P4JChangeList` interface.

Because Perforce file operations can typically run to a conclusion even with errors or warnings caused by incoming arguments, most file-related methods do not throw exceptions when a request error is encountered. Instead, the file-related methods return a Java list of results, which can be scanned for errors, warnings, informational messages, and the successful file specs normally returned by the server. P4Java provides helper classes and methods to detect these errors.

P4Java file methods are also designed to be composable: the valid output of one file method (for instance, `P4JServer.getDepotFileList`) can usually be passed directly to another file method (such as `P4JClient.editFiles`) as a parameter. This approach can be very convenient in complex contexts such as ant or Eclipse plug-ins, which perform extensive file list processing.

The following example shows you how to get and process a simple list of all depot files corresponding to the local directory (assuming a client is associated with the server):

```
String[] filePaths = new String[] {".."};
List<P4JFileSpec> fileList = server.getDepotFileList(
    P4JFileSpecBuilder.makeFileSpecList(filePaths)
);
for (P4JFileSpec file : fileList) {
    if (file.getOpStatus() == P4JFileSpecOpStatus.VALID) {
        // Valid file spec; do something with it... (you would also
        // normally check the paths for null in production contexts).
        System.out.println(file.getDepotPath());
        System.out.println(file.getClientPath());
        System.out.println(file.getLocalPath());
    } else {
        // Error - print the message:
        System.err.println(file.getStatusMessage());
    }
}
```

An example of composition (again, assuming valid client and server objects):

```
List<P4JFileSpec> fileList = client.openedFiles(
    P4JFileSpecBuilder.makeFileSpecList(
        new String[] {"//depot/.."},
        0, P4JChangeList.DEFAULT));
List<P4JExtendedFileSpec> extendedList = server.getExtendedFileList(
    P4JFileSpecBuilder.getValidFileSpecs(fileList),
    ... // other method options here...
);
for (P4JExtendedFileSpec extendedFSpec : extendedList) {
    if (extendedFSpec.getOpStatus() == P4JFileSpecOpStatus.VALID) {
        // Do something with the extended file spec information...
    }
}
```

Advanced usage notes

The following notes provide guidelines for developers using features beyond the basic usage model.

Perforce server addresses, URIs, and properties

P4Java uses a URI string format to specify the network location of target Perforce SCM servers. This URI string format is described in detail in the server factory documentation, but it always includes at least the server's hostname and port number, and a scheme part that indicates a P4Java connection (for example, `p4java://localhost:1666`). Note that:

- P4Java does not obtain default values from the execution environment or other sources for any part of the URI string. All non-optional parts of the URI must be filled in. (For example, P4Java does not attempt to retrieve the value of `P4PORT` from a Unix or Linux environment to complete a URL with a missing port number.)
- P4Java's factory methods allow you to pass properties into the `P4JServer` object in the server's URI string as query parts that override any properties that are passed in through the normal properties parameter in the server factory `getServer` method. This feature is somewhat limited in that it doesn't currently implement URI escape sequence parsing in the query string, but it can be very convenient for properties passing. See "P4Java properties" on page 13 for an explanation.

The `P4JServerResource` Interface

P4Java represents Perforce server objects (such as changelists, branch specs, job specs, and so on) to the end user through associated interfaces (such as `P4JChangeList`, `P4JBranchSpec`, and so on) onto objects within P4Java that mirror or proxy the server-side originals. This means that over time, the P4Java-internal versions of the objects may get out of date with the server originals, or the server originals may need to be updated with corresponding changes made to the P4Java versions.

P4Java's `P4JServerResource` interface is designed to support such proxying and to allow refreshes from the server or updates to the server as necessary. Virtually all useful P4Java objects or interfaces that proxy or represent Perforce server-side objects extend the `P4JServerResource` interface, and unless otherwise noted in the individual Javadoc comments, the interface methods can be used to update server- and client-side versions accordingly.

P4Java properties

P4Java uses Java properties to set various operational values for specific P4JServer instances and/or for P4Java as a whole. These properties are typically used for things like preferred temporary file directory locations, application version and name information for Perforce server usage, and the location of a suitable Perforce authentication tickets file (see “Authentication” on page 17 for details). A full list of publicly-visible properties (with default values) is given in the P4JProperties Javadoc.

Properties intended for P4Java use can have “long form” or “short form” names. Long form names are canonical, and are always prefixed by the string represented by `P4JProperties.P4JAVA_PROP_KEY_PREFIX` (normally `com.perforce.p4java.`, for example, `com.perforce.p4java.userName`). Short form names are the same string without the standard prefix (for example, `userName`). Use long form names when there’s any chance of conflict with system or other properties; short form names, on the other hand, are convenient for putting property values onto URI strings as long as you know the property name won’t collide with another property name in use by the app or system.

Properties can be passed to P4Java in several ways:

- As properties originally passed to the JVM using the usual Java JVM and system properties mechanisms.

Passing properties in this way is useful for fundamental P4Java-wide values that do not change over the lifetime of the P4Java invocation and that do not differ from one P4JServer instance to another. A typical example of such a property is the `com.perforce.p4java.tmpDir` property, which is used by P4Java to get the name of the temporary directory to be used for P4Java tmp files (and which defaults to `java.io.tmpdir` if not given).

- As properties passed in to an individual P4JServer instance through the server factory `getServer` method’s `properties` parameter.

Properties passed in this way override properties passed in through the JVM. This mechanism is useful for any properties that are (or may be) server-specific, such as `userName`, `clientName`, and so on.

- As properties passed in through the server factory’s URI string parameter query string.

Properties passed in this way override properties passed in through the `properties` parameter and the JVM. This mechanism is useful for ad hoc property-passing and/or overriding less-changeable properties passed in through the `properties` parameter.

The following code shows an example of passing properties to a P4JServer instance using the URI string query mechanism:

```
P4JServer server = P4JServerFactory.getServer(  
    "p4java://test:1666?userName=test12&clientName=test12_client&autoConnect=y", null);
```

Assuming no errors occur along the way, this code returns a P4JServer object connected to the Perforce server host `test` on port 1666 with the Perforce client name `test12_client` and Perforce user name `test12` logged in automatically (note that the login only works if the underlying authentication succeeds - see “Authentication” on page 17 for details).

Character Set Support

Character set support is only enabled for Unicode-enabled Perforce servers. In this mode, P4Java differentiates between Perforce file content character sets (that is, the encoding used to read or write a file’s contents) and the character sets used for Perforce file names, job specs, changelist descriptions, and so on.

This distinction is made due to the way Java handles strings and basic I/O: in general, while file content character set encodings need to be preserved so that the end results written to or read from the local disk are properly encoded, P4Java does not need to know about file metadata or other string value encodings. Because Perforce servers store and transmit all such metadata and strings in normalized UTF-8 form, and because all Java strings are inherently encoded in UTF-16, the encoding to and from non-UTF-16 character sets (such as `shiftjis`) is done externally from P4Java (usually by the surrounding app), and is not influenced by or implemented in P4Java itself. This means that the character set passed to the `P4JServer.setCharsetName` method is only used for translation of file content. Everything else, including all file names, job specs, changelist descriptions, and so on, is encoded in the Java-native Java string encoding UTF-16 (and may or may not need to be translated out of that coding to something like `shiftjis` or `winansi`).

P4Java supports file content operations on files encoded in most of the character sets supported by the Perforce server, but not all. The list of supported Perforce file content charsets is available to calling programs through the `P4JServerCharsets.getKnownCharsets` method. If you attempt to set a P4JServer object’s charset to a charset not supported by both the Perforce server and the local JDK installation, you will get an appropriate exception; similarly, if you try to (for example) sync a file with an unsupported character set encoding, you will also get an exception.

The Perforce server uses non-standard names for several standard character sets. P4Java also uses the Perforce version of the character set, rather than the standard name.

Error Message Localization

Error messages originating from the Perforce server are localized if the Perforce server is localized; error messages originating in P4Java itself are not currently localized. P4Java's internal error messages aren't intended for end-user consumption as-is, and your applications should process these errors into localized versions for presentation to end users.

Logging and tracing

P4Java includes a simple logging callback feature, documented in the `P4JLogCallback` Javadoc page, that enables consumers to log P4Java-internal errors, warnings, informational messages, exceptions, and so on. This feature performs no message formatting or packaging. You can put the log message through the surrounding application context's logger as required. In general, your applications should log all error and exception messages. Informational messages, statistics, and warning messages do not need to be logged unless you are working with Perforce support to debug an issue.

P4Java also includes an extensive tracing callback feature. This feature is useful for debugging, but it is verbose and can fill log files very quickly.

Exceptions and errors

To signal recoverable Perforce errors, P4Java uses a fairly standard set of extensions to `java.lang.Exception`. However, P4Java also uses extensions to `java.lang.Error` to signal unrecoverable errors that it detected itself (such as null pointers or out-of-range values).

Because these extensions are defined as extensions to `Error`, they're not typically declared in method "throws" clauses. Your applications can ignore them in the same way that they can ignore non-P4Java `Error` throwables. It is good practice, however, to keep an outer catch block for `P4JError` errors, as well as other exceptions and throwables. P4Java throws such errors only for conditions that show a serious programming error.

Standard implementation classes

The `com.perforce.p4java.impl.generic` package is the root for a fairly large set of standard implementation classes such as `P4JJJob`, `P4JChangelist`, and so on. These implementation classes are used internally by P4Java, and while usage is not mandatory, you are encouraged to use them as well.

I/O and file metadata issues

The quality of P4Java's network and file I/O in real-world usage is strongly affected by the quality of implementation of the underlying Java NIO packages. Java's handling of file metadata also affects I/O. Although JDK 6 is an improvement over JDK 5, it can be

difficult to manipulate file type and metadata (such as permissions, access/modification time, symlinks, and so on) in pure Java. These are abilities that C programmers take for granted. Issues typically arise from JVM limitations such as an inability to set read-only files as writable, reset modification times, observe Unix and Linux umask values, and so on.

Because of these issues, P4Java has a file metadata helper callback scheme, defined in `com.perforce.p4java.impl.generic.sys.P4JSystemFileCommandsHelper`. This approach enables users to register their own file metadata class helper (typically using something like an Eclipse file helper or a set of native methods) with the server factory, to help in cases where the JDK is not sufficient. See the relevant Javadoc for details.

Threading issues

P4Java is inherently thread-safe when used properly. The following best practices can help to ensure that users do not encounter thread-related problems:

- P4Java's `P4JServer` object is partially thread-safe. The only state preserved in the underlying implementation classes is the Perforce client that is associated with the server, and the server's authentication state.
- You can have multiple threads working against a single `P4JServer` object simultaneously, but note that changing authentication state (login state, password, user name, and so on) or the client that is associated with the server can have unpredictable results on long-running commands that are still running against that server object. You should ensure that changing these attributes only happens when other commands are not in progress with the particular server object.
- P4Java makes no guarantees about the order of commands sent to the Perforce server by your applications. You must ensure that any required ordering is enforced.
- Using a large numbers of threads against a single `P4JServer` object can impose a heavy load on the JVM and the corresponding server. To control load, create your own logic for limiting thread usage. Be certain that your use of threads does not cause deadlock or blocking. Consider using a single `P4JServer` object for each thread.
- P4Java offers a number of useful callbacks for things like logging, file helpers, progress monitoring, and so on. These callbacks are performed within a live thread context. Ensure that any callbacks that you register or use do not cause blocking or deadlocks.
- To obtain the best resource and memory allocation strategies for your specific threading needs, experiment with JVM invocation parameters. Garbage collection and memory allocation strategies can make quite a difference in raw threading throughput and latency, but often indirectly and unpredictably.

Authentication

P4Java implements both the Perforce tickets-based authentication and the Perforce single sign on (SSO) feature. Both types of authentication are described in detail in the P4Java Javadoc, but some P4Java-specific issues to note include:

- P4Java manages a `p4 tickets` file in a manner similar to that of the P4 command line (under normal circumstances, the two can share the same tickets file). When a ticket value is requested by the Perforce server and the current ticket value in the associated P4JServer object is not set, an attempt will be made to retrieve the ticket out of the `p4 tickets` file. If found, the ticket is stored on the P4JServer object and used as the Perforce authentication ticket.

A successful login causes the ticket value to be added or updated in the tickets file, and a logout causes the current ticket value in the `p4 tickets` file to be removed. The P4JServer object's ticket should be set to `null` to cause a re-reading of the ticket value from the `p4 tickets` file.

The `p4 tickets` file is usually stored in the same place the `p4` command line stores it, but the `P4JProperties.TICKET_PATH_KEY` property can be used to specify an alternate tickets file.

- P4Java implements Perforce's SSO scheme using a callback interface described in the P4JSSOCallback Javadoc (in the package `com.perforce.p4java.server.callback`). Ensure that the callback doesn't block, and that it adheres to the expected format of the associated Perforce server.

Other Notes

- As documented in the main Perforce documentation, Perforce form triggers can cause additional output on form commands such as "change" or "client", even when the trigger succeeds. This trigger output is available through the P4Java command callback feature, but note that there is currently no way to differentiate trigger output from normal command output, and that such trigger output will also be prepended or appended to normal string output on commands such as `P4JServer.newLabel`.
- P4Java's command callback feature, documented in class `com.perforce.p4java.server.callback.P4JCommandCallback`, is a useful way to get blow-by-blow command status messages and trigger output messages from the server in a way that can mimic the `p4` command line client's output. Usage is straightforward, but note the potential for deadlocks and blocking if you are not careful with callback method implementation.

- P4Java's progress callback feature gives users a somewhat impressionistic measure of command progress for longer-running commands. Progress callbacks are documented in the Javadoc for class `com.perforce.p4java.server.callback.P4JProgressCallback`. Once again, if you use this feature, ensure that your callback implementations do not cause deadlocks or blocking.
- We strongly recommend setting the `progName` and `progVersion` properties (either globally or for each P4JServer instance) whenever you use P4Java. Set these values to something meaningful that reflects the application or tool in which P4Java is embedded; this can help Perforce administrators and application debugging.

For example, the following code sets `progName` and `progVersion` via the JVM invocation property flags:

```
java -Dcom.perforce.p4java.programName=p4test
     -Dcom.perforce.p4java.programVersion=2.01A ...
```

Alternatively, you can also use the server factory `getServer` method's properties parameter:

```
Properties props = new Properties(System.getProperties());
props.setProperty(P4JProperties.PROG_NAME_KEY, "ant-test");
props.setProperty(P4JProperties.PROG_VERSION_KEY, "Alpha 0.9d");
...
server = P4JServerFactory.getServer(serverUri, props);
```

- If your application receives a `P4JConnectionException` from a `P4JServer` or `P4JClient` method while communicating with a Perforce server, the only safe action is to close the connection and start over with a new connection, rather than continue using the connection.

A `P4JConnectionException` event typically represents a serious network error (such as the Perforce server unexpectedly closing a connection or a bad checksum in a network packet), and there's no guarantee that after receiving such an event the connection is even usable, let alone reliable.

- There is currently no `diff` method on `P4JFileSpec` interfaces to compare versions of the same Perforce-managed file, but this functionality may be easily implemented with a combination of `P4JServer.getFileContents` to retrieve the contents of specific

versions to temporary files, and the use of the operating system's diff application on these temporary files as shown below:

```

InputStream fspecStream1 = server.getFileContents(
    P4JFileSpecBuilder.makeFileSpecList(
        new String[] {spec1}), false, true);
InputStream fspecStream2 = server.getFileContents(
    P4JFileSpecBuilder.makeFileSpecList(
        new String[] {spec2}), false, true);
File file1 = null;
File file2 = null;
try {
    file1 = File.createTempFile("p4jdiff", ".tmp");
    file2 = File.createTempFile("p4jdiff", ".tmp");
    FileOutputStream outputStream1 = new FileOutputStream(file1);
    FileOutputStream outputStream2 = new FileOutputStream(file2);
    byte[] bytes = new byte[1024];
    int bytesRead = 0;
    while ((bytesRead = fspecStream1.read(bytes)) > 0) {
        outputStream1.write(bytes, 0, bytesRead);
    }
    fspecStream1.close();
    outputStream1.close();
    while ((bytesRead = fspecStream2.read(bytes)) > 0) {
        outputStream2.write(bytes, 0, bytesRead);
    }
    fspecStream2.close();
    outputStream2.close();

    Process diffProc = Runtime.getRuntime().exec(new String[] {
        "/usr/bin/diff", file1.getPath(), file2.getPath()
    });

    diffProc.waitFor();
    if (diffProc != null) {
        InputStream iStream = diffProc.getInputStream();
        byte[] inBytes = new byte[1024];
        int inBytesRead = 0;
        while ((inBytesRead = iStream.read(inBytes)) > 0) {
            System.out.write(inBytes, 0, inBytesRead);
        }
    }
} catch (Exception exc) {
    P4Cmd.error("diff error: " + exc.getLocalizedMessage());
    return;
} finally {
    if (file1 != null) file1.delete();
    if (file2 != null) file2.delete();
}

```

