HelixCore

P4Java API User Guide

2017.2

October 2017

PERFORCE

www.perforce.com

Contents

How to use this guide

This guide provides information on installing and using P4Java, and assumes a basic knowledge of both Java (JDK 5 or later) and Helix Server.

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other documentation

See https://www.perforce.com/support/self-service-resources/documentation.

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
[`-f`]	The enclosed elements are optional. Omit the brackets when you compose the command.
...	■ Repeats as much as needed: • `alias-name[[$(arg1)... [$(argn)]]=transformation` ■ Recursive for all directory levels: • `clone perforce:1666 //depot/main/p4... ~/local-repos/main` • `p4 repos -e //gra.../rep...`
element1 | *element2*	Either *element1* or *element2* is required.

P4Java Programming

Perforce Software's P4Java is a Java API that enables applications to access Helix Server, the Perforce enterprise version control system in a "Java natural" and Java-native way. P4Java presents Helix Server services and 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 Helix Server 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 Helix Server services from within the application
- Plug-ins for Java tools such as Eclipse, ant, Mylyn, Cruise Control, and so on, that need to communicate with Helix Server
- 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

System Requirements

P4Java assumes the presence of a JDK 6 or later environment, but will work against a JDK 5 installation, with some limitations.

Due to current US export control restrictions for some countries, the standard JDK package only comes with 128 bit encryption level cyphers. In order to use P4Java to connect to an SSL-enabled Helix Server, those living in eligible countries may download the unlimited strength JCE (Java Cryptography Extension) package and replace the current default cryptography jar files with the unlimited strength files.

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:

P4Java API Javadoc

Sample programs

Sample P4Java applications are available in Helix Server's public depot.

To access the public depot, set `P4PORT` to `public.perforce.com:1666` and add the depot path `//guest/perforce_software/p4java/samples/basic/...` to your client workspace view.

These samples are used throughout this document to illustrate common usage patterns and simple code snippets, and can also be used as the basis for further user experiments with P4Java development.

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 `IServer` server interface, and associated classes and interfaces related to the `IServer` definition. This package enables participating applications to connect to Helix Servers and start interacting with Helix Server services through the `IServer` interface.
- `com.perforce.p4java.client`: defines the `IClient` client interface and associated classes and support definitions. Participating applications typically use the IClient interface to access Helix Server 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 Helix Server-managed objects such as changelists, jobs, and clients.
- `com.perforce.p4java.core.file`: contains the main `IFileSpec` interface for accessing and defining the various types of files that Helix Server 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 Helix Server 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 Helix Server usage:

1. A Java application uses the P4Java `ServerFactory` class to obtain a `IServer` interface onto a specific Helix Server at a known network address and port, and connects to this Helix Server through the `IServer` interface that is returned from the factory.

2. The application optionally logs in to the Helix Server through the `IServer`'s login and associated methods.
3. The application obtains a suitable `IClient` interface into a Helix Server client workspace through the `IServer` interface's "get client" method.
4. The application syncs the Helix Server client workspace through the `IClient` 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 Helix Server client workspace, through the `IServer` and `IClient` interfaces.
6. The application adds, edits, or deletes files in the local Helix Server client workspace using the `IClient` interface. These files are added to the default or a numbered Helix Server changelist represented by one or more `IChangeList` interfaces, which are obtained through the `IClient` or `IServer` 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 `IChangeList` interface. This submission can be linked with one or more Helix Server jobs, represented by the `IJob` interface.
8. The application can switch between Helix Server workspaces, browse Helix Server jobs and changelists, log in as a different user, and add, edit, or delete files, using the relevant `IServer` or `IClient` interfaces.
9. To disconnect from a Helix Server, the application calls the `disconnect` method on the `IServer` interface.

This usage model relies heavily on representing all significant Helix Server 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 Helix Server errors as Java exceptions that are subclassed out of the main `P4JException` class, and thrown from nearly every significant `IServer` and `IClient` 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 Helix 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 "Exception and error handling" on page 10.

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 "Helix Server file operations" on page 11.

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

Unlike the Helix C/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 Helix Server 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 `IServer` 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 18.

Typical usage patterns

This section briefly describes typical usage patterns and provides a starting point for developers using P4Java for the first time. Many examples below are snippets from (or refer to) the P4Java sample programs available in the Helix Server public depot.

To access the public depot, set `P4PORT` to `public.perforce.com:1666` and add the depot path `//guest/perforce_software/p4java/samples/basic/...` to your client workspace view.

The IServer and IClient interfaces and the ServerFactory class

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

The snippet below is from the `ServerFactoryDemo` class in the sample package, and shows a very simple way to prompt the user for a Helix Server URI, connect to the server at the URI, and get basic information about that server. This is the basic “Hello World!” P4Java application, and works like the **p4 info** command (with suitable attention being paid to formatting details with the `formatInfo` method below).

```
BufferedReader lineReader = new BufferedReader(
                            new InputStreamReader(System.in));
try {
  for (;;) {
    System.out.print(PROMPT);
    String serverUriString = lineReader.readLine();
    if ((serverUriString == null) ||
        serverUriString.equalsIgnoreCase("quit")) {
      break;
```

```
    } else {
      IServer server = ServerFactory.getServer(serverUriString, null);
      server.connect();
      IServerInfo info = server.getServerInfo();
      if (info != null) {
        System.out.println(
          "Info from Perforce server at URI '"
          + serverUriString + "':");
        System.out.println(formatInfo(info));
      }
      if (server != null) {
        server.disconnect();
      }
    }
  }
} catch (RequestException rexc) {
  System.err.println(rexc.getDisplayString());
  rexc.printStackTrace();
} catch (P4JavaException exc) {
  exc.printStackTrace();
} catch (IOException ioexc) {
  ioexc.printStackTrace();
} catch (URISyntaxException e) {
  e.printStackTrace();
}
```

Multiple `IServer` objects can represent the same physical Helix 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 16.

Helix Server client workspaces are represented by the `com.perforce.p4java.client.IClient` interface, which can be used to issue Helix Server client workspace-related commands such as sync commands, file add /delete / edit commands, and so on. A `IClient` interface is typically obtained from an `IServer` interface using the `getClient()` method, and is associated with the `IServer` using the `setCurrentClient()` method as illustrated in the `ClientUsageDemo` snippet below:

```
IServer server = null;
  try {
    server = getServer(null);
    server.setUserName(userName);
    server.login(password);
    IClient client = server.getClient(clientName);
    if (client != null) {
      server.setCurrentClient(client);
      // use the client in whatever way needed...
    }
  } catch (Exception exc) {
    // handle errors...
  }
```

Note also the use of the `setUserName` and `login` methods on the server to establish the current user and log them in, respectively.

Note also, that unlike the `p4` command line client, there are no defaults for user and workspace. Your application must explicitly associate a workspace (an `IClient` client object) and user with the server using the `IServer.getClient` and `IServer.setCurrentClient` methods.

Exception and error handling

P4Java uses a small set of Java exceptions to signal errors that have occurred in either the Helix Server as a result of issuing a specific command to the server, or in the P4Java plumbing in response to things like TCP/IP connection errors or system configuration issues. (These exceptions are not used to signal file operation problems at the individual file level — see "Helix Server file operations" on the facing page for details about individual file error handling.)

In general, P4Java exceptions are rooted in two different classes: the `P4JavaException` classes are intended for “normal” (that is, recoverable) errors that occur as the result of things like missing client files, a broken server connection, or an inappropriate command option; the `P4JavaError` classes are intended for more serious errors that are unlikely to be recoverable, including unintended null pointers or P4Java-internal errors. The `P4JavaException` class hierarchy is rooted in the normal `java.lang.Exception` tree, and any such exception is always declared in relevant method “throws” clauses; the `P4JavaError` classes, however, are rooted in `java.lang.Error`, and consequently do not need to be declared or explicitly caught. This allows a developer to catch all such `P4JavaError`s, for example, in an outer loop, but to process “normal” `P4JavaException`s in inner blocks and loops as they occur.

Typically, application code should report a `P4JavaError` exception and then terminate either itself or whatever it was doing as soon as possible, as this exception indicates a serious error within P4Java. `P4JavaException` handling is more fine-grained and nuanced: A `P4JavaException` almost always signals a recoverable (or potentially-recoverable) error, and should be caught individually or at the class level. The following snippet represents a common pattern for `P4Java` error and exception handling around major functional blocks or processing loops:

```
try {
  // issue one or more server or client commands...
} catch (P4JavaError err) {
  panic(err); // causes app to exit after printing message to stderr...
} catch (RequestException rexc) {
  // process server-side Perforce error...
} catch (ConnectionException cexc) {
  // process Perforce connection exception...
} catch (P4JavaException exc) {
  // catchall...
} catch (Exception exc) {
  // Other-exception catchall...
}
```

Note the way `RequestException` and `ConnectionException` events are handled separately: `RequestException` exceptions are almost always thrown in response to a Helix Server error message and therefore include a severity and generic code that can be used or displayed (other `P4JavaExceptions` do not usually contain these), and `ConnectionExceptions` should normally result in the enclosing app explicitly closing or at least re-trying the associated connection, as processing can no longer continue on the current Helix Server connection.

Helix Server file operations

To define common Helix Server-managed file attributes and options, P4Java uses the `com.perforce.p4java.core.file.IFileSpec` 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 Helix Server file-related methods are available on the `IServer` and `IClient` interfaces, and might also be available on other interfaces such as the `IChangeList` interface.

Because Helix Server file operations can typically run to a conclusion even with errors or warnings caused by incoming arguments, and because the server usually interpolates error and informational messages in the list of file action results being returned, 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, `IServer.getDepotFileList`) can usually be passed directly to another file method (such as `IClient.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 snippet below, from the sample `ListFilesDemo` class, illustrates a very common pattern used when retrieving a list of files (in this case from the `getDepotFiles` method):

```
List<IFileSpec> fileList = server.getDepotFiles(
      FileSpecBuilder.makeFileSpecList(new String[] {"//..."}), false);
  if (fileList != null) {
    for (IFileSpec fileSpec : fileList) {
      if (fileSpec != null) {
        if (fileSpec.getOpStatus() == FileSpecOpStatus.VALID) {
          System.out.println(formatFileSpec(fileSpec));
        } else {
          System.err.println(fileSpec.getStatusMessage());
        }
      }
    }
  }
```

Note in particular the use of the `FileSpecBuilder.makeFileSpecList` helper method that converts a String array to a list of `IFileSpec` objects; note also the `formatFileSpec` method referenced above; this simply prints the depot path of the returned `IFileSpec` object if it's valid.

Summary vs. Full Objects

The 2009.2 release of P4Java introduced the notion of "summary" and "full" representations of objects on a Helix Server. In many cases, the Helix Server only returns summaries of objects that it's been asked to list. For example, if you issue a `p4 clients` command to a server, what comes back is a list of client metadata for known client workspaces, but not the associated workspace views. For things like changelists, jobs, branches, and so on, to obtain the full version of the Helix Server object (such as a specific client workspace), you typically do a `p4 client -o` with the workspace's name.

Similarly, P4Java distinguishes between the summary objects returned from the main list methods (such as `IServer.getClients()`) and the full objects returned from individual retrieval methods (such as `IServer.getClient()`).

The snippet below, edited from the `ListClientDemo` sample app, illustrates a typical usage pattern for summary and full object retrieval:

```
try {
  IServer server = getServer(null);
  server.setUserName(userName);
```

```
  server.login(password);
  List<IClientSummary> clientList = server.getClients(
                                        userName, null, 0);
  if (clientList != null) {
    for (IClientSummary clientSummary : clientList) {
      // NOTE: list returns client summaries only; need to get
      // the full client to get the view:
      IClient client = server.getClient(clientSummary);
      System.out.println(client.getName() + " "
          + client.getDescription().trim() + " "
          + client.getRoot());
      ClientView clientView = client.getClientView();
      if (clientView != null) {
        for (IClientViewMapping viewMapping : clientView) {
          System.out.println("\t\t" + viewMapping);
        }
      }
    }
  }
} catch (RequestException rexc) {
  System.err.println(rexc.getDisplayString());
  rexc.printStackTrace();
} catch (P4JavaException exc) {
  exc.printStackTrace();
} catch (URISyntaxException e) {
  e.printStackTrace();
}
```

Note that only clients owned by *username* are returned, and that in order to print the associated client workspace view for each retrieved summary client workspace, we get the full client object. This is more common in cases where a user might iterate across a list of all workspaces known to the Helix Server in order to find a specific client workspace, then retrieve that client (and only that client) workspace in full.

Advanced usage notes

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

Helix Server addresses, URIs, and properties

P4Java uses a URI string format to specify the network location of target Helix 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 `IServer` 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 the facing page for an explanation.

SSL connection support

Helix Server at release 2012.1 or higher supports 256-bit SSL connections and trust establishment by accepting the fingerprint of the SSL certificate's public key.

Due to current US export control restrictions for some countries, the standard JDK package only comes with 128 bit encryption level cyphers. In order to use P4Java to connect to an SSL-enabled Helix Server, those living in eligible countries may download the unlimited strength JCE (Java Cryptography Extension) package and replace the current default cryptography jar files with the unlimited strength files.

To make a secure connection using P4Java, append `ssl` to the end of the P4Java protocol (for example, `p4javassl://localhost:1667`). For a new connection or a key change, you must also (re)establish trust using the `IOptionsServer`'s `addTrust` method. For example:

```
// Create a P4Java SSL connection to a secure Perforce server
try {
  String serverUri = "p4javassl://localhost:1667";
  Properties props = null;
  IOptionsServer server = ServerFactory.getOptionsServer(serverUri,
                          props);

  // assume a new first time connection
  server.addTrust(new TrustOptions().setAutoAccept(true));

  // if all goes well...
```

```
  IServerInfo serverInfo = server.getServerInfo();
} catch (P4JavaException e) {
  // process P4Java exception
} catch (Exception e) {
  // process other exception
}
```

The IServerResource Interface

P4Java represents Helix Server objects (such as changelists, branch mappings, job specs, and so on) to the end user through associated interfaces (such as `IChangeList`, `IBranchSpec`, 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 `IServerResource` 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 Helix Server-side objects extend the `IServerResource` 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 `IServer` 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 Helix Server usage, and the location of a suitable Helix Server authentication tickets file (see "Authentication" on page 18 for details). A full list of publicly-visible properties (with default values) is given in the `PropertyDefs` 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 `PropertyDefs.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 `IServer` 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 `IServer` 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 `IServer` instance using the URI string query mechanism:

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

Assuming no errors occur along the way, this code returns a `IServer` object connected to the Helix Server host `test` on port 1666 with the Helix Server client name `test12_client` and Helix Server user name `test12` logged in automatically (note that the login only works if the underlying authentication succeeds — see "Authentication" on page 18 for details.

Character Set Support

Character set support is only enabled for Unicode-enabled Helix Servers. In this mode, P4Java differentiates between Helix Server file content character sets (that is, the encoding used to read or write a file's contents) and the character sets used for Helix Server 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 Helix Server 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 `IServer.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 Helix Server, but not all. The list of supported Helix Server file content charsets is available to calling programs through the `PerforceCharsets.getKnownCharsets` method. If you attempt to set a `IServer` object's charset to a charset not supported by both the Helix 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 Helix Server uses non-standard names for several standard character sets. P4Java also uses the Helix Server version of the character set, rather than the standard name.

Error Message Localization

Error messages originating from the Helix Server are localized if the Helix 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; your applications should process such errors into localized versions that are presentable to end users.

Logging and tracing

P4Java includes a simple logging callback feature, documented in the `ILogCallback` Javadoc page, that enables consumers to log P4Java-internal errors, warnings, informational messages, exceptions, and so on. Logging is enabled or disabled on a P4Java-wide basis, not on a per-connection or per-server basis.

The logging 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 Technical Support to debug an issue.

Standard implementation classes

The `com.perforce.p4java.impl.generic` package is the root for a fairly large set of standard implementation classes such as `Job`, `Changelist`, 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. This is especially useful if you wish to extend standard P4Java functionality by, for example, adding audit or authentication methods to standard classes.

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 often 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.ISystemFileCommandsHelper`. 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 `ISystemFileCommandsHelper` 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 `IServer` object is partially thread-safe. The only state preserved in the underlying implementation classes is the Helix Server client that is associated with the server, and the server's authentication state.
- You can have multiple threads working against a single `IServer` 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 Helix Server by your applications. You must ensure that any required ordering is enforced.
- Using a large numbers of threads against a single `IServer` 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 `IServer` 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 Helix Server tickets-based authentication and the Helix Server 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 matter 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 Helix Server and the current ticket value in the associated `IServer` object is not set, an

attempt is made to retrieve the ticket out of the `p4 tickets` file. If found, the ticket is stored on the `IServer` object and used as the Helix Server 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 `IServer` 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 `PropertyDefs.TICKET_PATH_KEY` property can be used to specify an alternate tickets file.

- P4Java implements Helix Server's SSO scheme using a callback interface described in the `ISSOCallback` 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 Helix Server.

Other Notes

- As documented in the main Helix Server documentation, Helix Server 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 `IServer.newLabel`.
- P4Java's command callback feature, documented in class `com.perforce.p4java.server.callback.ICommandCallback`, 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.IProgressCallback`. 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 `IServer` 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 Helix Server 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(PropertyDefs.PROG_NAME_KEY, "ant-test");
props.setProperty(PropertyDefs.PROG_VERSION_KEY, "Alpha 0.9d");

    ...

server = IServerFactory.getServer(serverUriString, props);
```

- If your application receives a `ConnectionException` from a `IServer` or `IClient` method while communicating with a Helix Server, the only truly safe action is to close the connection and start over with a new connection, rather than continue using the connection.

  A `ConnectionException` event typically represents a serious network error (such as the Helix 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 `IFileSpec` interfaces to compare versions of the same Helix Server-managed file, but this functionality may be easily implemented with a combination of `IServer.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(
  FileSpecBuilder.makeFileSpecList(
    new String[] {spec1}), false, true);
InputStream fspecStream2 = server.getFileContents(
  FileSpecBuilder.makeFileSpecList(
    new String[] {spec2}), false, true);

File file1 = null;
File file2 = null;

try {
  file1 = File.createTempFile("p4jdiff", ".tmp");
  file2 = File.createTempFile("p4jdiff", ".tmp");
  FileOutputStream outStream1 = new FileOutputStream(file1);
  FileOutputStream outStream2 = new FileOutputStream(file2);
  byte[] bytes = new byte[1024];
  int bytesRead = 0;
  while bytesRead = fspecStream1.read(bytes > 0) {
    outStream1.write(bytes, 0, bytesRead);
  }
  fspecStream1.close();
  outStream1.close();
  while bytesRead = fspecStream2.read(bytes > 0) {
    outStream2.write(bytes, 0, bytesRead);
  }
  fspecStream2.close();
  outStream2.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) {
  error("diff error: " + exc.getLocalizedMessage());
  return;
} finally {
  if (file1 != null) file1.delete();
  if (file2 != null) file2.delete();
}
```

License Statements

Perforce Software includes software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).