# RFC 0132 Command Line Interface and Launching

Draft

**41 Pages**

## Abstract

This RFC describes a proposed specification for a Command processing interface for the OSGi Framework.

# 0 Document Information

## 0.1 Table of Contents

## 0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 8.1.

`Source code is shown in this typeface.`

## 0.3 Revision History

The last named individual in this history is currently responsible for this document.

| Revision | Date | Comments |
|---|---|---|
| Initial | 05 MAR 2008 | Peter.Kriens@aQute.biz |
| Additional text | 03 APR 2008 | Added a large number of sections, mainly booting and more details about the processing of scripts. Also completely changed the API and added a problem description and requirements section. Changes are so massive that it was not that useful to track changes (and I forgot anyway) |
| Update | 15 MAY 2008 | More details in booting, minor update in the language aspects |
| Update | 31 JUL 2008 | Changed some method names, updated Javadoc and added to build |
| Update | 5 AUG 2008 | Updated javadoc section. Changed "boot" to "launch". |

# 1 Introduction

This RFC is an solution for RFP 99 Command Provider and RFP 80 Framework booting. This RFC outlines the interfaces necessary to implement command line shells in OSG frameworks as well as providing a generic start up solution for different implementations of frameworks.

# 2 Problem Description

This RFC addresses the problem of standardized external access. The purpose of this RFC is to allow any OSGi framework to launched, configured, and controlled externally from a console, telnet session, serial port or script. The framework must enable a set of basic commands that are supported by all implementations, but it must enable that bundles can provide additional commands.

## 2.1 Framework Launching

Every OSGi framework must invent its own technology for getting started. Therefore, code that may need to launch the framework from various places must write code that is proprietary for the particular OSGi vendors' implementation, if they wish to support more than one framework.

In enterprises, where the issues of vendor lock-in can cause a barrier to adoption of the system, this issue becomes magnified. While it should certainly be the case that each vendor can supply add-ons and extra features, the standard portions of all OSGi frameworks could be encapsulated in a Framework Launching specification. Having such a specification would increase the consistency and quality of compliant OSGi frameworks, and allay fears about vendor lock-in.

## 2.2 Command Interface

There is a need for a service that allows human users as well as well as programs to interact with on OSGi based system with a line based interface: a shell. This shell should allow interactive and string based programmatic access to the core features of the framework as well as provide access to functionality that resides in bundles.

Shells can be used from many different sources it is therefore necessary to have a flexible scheme that allows bundles to provide shells based on telnet, the Java 6 Console class, plain Java console, serial ports, files, etc.

Supporting commands from bundles should be made very lightweight and simple as to promote supporting the shell in any bundle. Commands need access to the input and output streams. Commands sometimes need to store state between invocations in the same session.

There is a need for a very basic shell functionality in small embedded devices, however, the design should permit complex shells that support background processing, piping, full command line editing, and scripting. It is possible that a single framework holds multiple shells.

The shell must provide a means to authenticate the user and the commands must be able to investigate the current user and its authorizations, preferably through standardized security mechanisms. To minimize footprint, it must also be possible to implement a shell without security.

# 3 Requirements

## 3.1 Non Functional

- Lightweight to allow shells for low footprint devices
- Allow shells with piping, background, scripting, etc.
- Make commands trivial to implement
- Make it easy to connect the shell to different sources.
- Provide an optional security framework based on existing security facilities
- Minimize the cost of a command (e.g. do not require eager loads of objects implementing commands)
- Support use of existing code by making a design that closely follows practices for command line applications in Java.

## 3.2 Launching

- The solution should allow for scripting languages to behave similarly for all compliant OSGi frameworks.
- The solution must allow for starting, restarting and stopping compliant OSGi frameworks without prior knowledge of the framework.
- The solution should allow for starting compliant frameworks in the same or different processes (though it would be OK to fall back on the Java Process verbs if necessary)
- The solution should handle the problems associated with launching multiple OSGi frameworks inside the same JVM process space.
- The solution must be able to handle vendor specific extensions.

## 3.3 Command Names

- Provide a list of basic command signatures to manage the framework so they are consistent among implementations.

## 3.4 Shell

- Provide interface to execute a string as command
- Allow other bundles to implement commands
- Allow other bundles to provide a connection to: telnet, console, serial port, etc.
- Provide help for each command
- Provide a means to disambiguate commands with the same name
- Provide a means to disambiguate when there are multiple shells
- Authenticate the user
- Provide programmatic access to the shell, that is, a program generates the commands.

## 3.5 Shell Commands

- Read input from user or previous command
- Write output to user or next command
- Allow sessions, i.e. group commands over a period of time, allowing them to share state
- Provide usage information of the command
- Allow commands to be protected with permissions
- Provide access to he authenticated user via User Admin (though User Admin may not be present)
- Optional: Allow computer readable meta information about the commands to support forms
- Optional: Provide formatting rules + library to standardize look and feel for output. This could consist of routines to show tables in a consistent form.
- Commands should not have to do low level parsing of command line arguments.
- Commands should be able to have access to the command line arguments
- Commands must be able to get access to the console input
- Commands must be able to use the keyboard input stream

## 3.6 Source Providers

- Provide an easy way to allow bundles to connect the shell to sources like telnet, serial ports, etc.

# 4 Technical Solution

## 4.1 Launching

The solution for the launch process is quite straightforward. Frameworks must designate a class that has an empty constructor and implements the following interfaces

- org.osgi.framework.Bundle
- org.osgi.framework.launch.SystemBundle (which extends Bundle)

The current specification provides a very detailed description of the System Bundle. The instantiated object represents this system bundle in an unstarted state (RESOLVED|INSTALLED). However, the system bundle must be able to provide a valid Bundle Context that can be used to install applications. These applications must not be started before the system bundle itself is started. The SystemBundle provides a number of methods that can be used to configure the framework **before** starting it. The System Bundle object can be configured, started, stopped, reconfigured, and started again, ad nauseum.

The System Bundle must implement the following methods.

- init(Properties) - The Properties object (which may be null is optionally backed by other Properties such a the System properties) must be used for the framework properties. The framework must use this properties as the only source by using getProperty (not get) so that the configurator can use the Properties linked behavior. If the properties object is null, useful defaults should be used to make the framework run appropriately in the current VM. I.e. the system packages for the current execution environment should be properly exported. Any persistent area should be defined in the current directory with a framework implementation specific name.

  The init method should be called before the start method is called. If the start method is called before the init method has been called, then start must call the init method with a null parameter. After the init method has been called, the system bundle must have a valid bundle context.

  A system bundle can be reinitialized by first stopping it and then calling init(Properties) again. Calling init when the system bundle is started must throw an Illegal State Exception.

  This method is not thread safe and must only be called from a single thread.

- waitForStop() - This method waits until the bundle is stopped and completely finished with the cleanup. This method will wait until someone calls, or has called, stop. If the system is not ACTIVE, then this method returns directly. This method is thread safe and can be called by different threads.

The following is a short example of creating a Felix framework, adding bundles to it, launching the framework and then waiting for it to stop. This assumes that some bundle (for example a shell) will initiate the stop command to the system bundle.

- `f = new org.apache.felix.Felix`
- `f init`
- `context = f bundleContext`
- `$context installBundle http://www.aQute.biz/repo/aQute/sample.jar`

- `f start`
- `f waitForStop`

This example makes the main thread wait for the framework to finish.

## 4.2 Life Cycle Issues

A framework must never do System.exit(n) when stopped. It is the responsibility of the configurator to exit.

A framework can be make repeated start/stop cycles. The semantics of stopping a framework are described in the core specification. It is not possible to change the properties. The options in the start and stop methods are ignored.

### what happens with update?

## 4.3 Properties

The configurator can set the following properties. In addition, it can also add framework implementation dependent properties.

| | |
|---|---|
| org.osgi.framework.version | set by framework implementation |
| org.osgi.framework.vendor | set by framework implementation |
| org.osgi.framework.language | set by configurator, but framework should provide a default when not set |
| org.osgi.framework. executionenvironment | set by configurator, but framework should provide a default when not set |
| org.osgi.framework.processor | set by configurator, but framework should provide a default when not set |
| org.osgi.framework.os.version | set by configurator, but framework should provide a default when not set |
| org.osgi.framework.os.name | set by configurator, but framework should provide a default when not set |
| org.osgi.supports. framework.extension | set by framework |
| org.osgi.supports. bootclasspath.extension | set by framework |
| org.osgi.supports.framework. fragment | set by framework to true |
| org.osgi.supports.framework. requirebundle | set by framework to true |
| org.osgi.framework. bootdelegation | set by configurator, framework provides empty default. |
| org.osgi.framework.system. packages | set by configurator, but framework should provide a default when not set |
| | |
| org.osgi.framework.security | The name of a Security Manager class with public empty constructor. A valid value is also true, this means that the framework should instantiate its own security manager. If not set, security could be defined by a parent framework or there is no security. This can be detected by looking if there is a security manager set |

| | |
|---|---|
| | ### ??? |
| org.osgi.framework.storage | A valid file path in the file system to a directory that exists. The framework is free to use this directory as it sees fit. This area can not be shared with anything else. If this property is not set, the framework should use a file area from the parent bundle. If it is not embedded, it must use a reasonable platform default. |
| org.osgi.framework.libraries | A list of paths (separated by path separator) that point to additional directories to search for platform specific libraries |
| org.osgi.framework.command.execpermission | The command to give a file executable permission. This is necessary in some environments for running shared libraries. |
| org.osgi.framework.root.certificates | Points to a directory with certificates. ###??? Keystore? Certificate format? |
| org.osgi.framework.windowsystem | Set by the configurator but the framework should provide a reasonable default. |

## 4.4 Starting Procedure

This RFC does not propose a proper shell script for the launch procedure. The Bundle object approach to framework creation is sufficient to allow almost any Java compatible script language to be used. An example of such a script language is the shell as proposed in this document (tsl), but any script language will work: Jython, Jruby, Beanshell, Bex, Groovy, etc.

For example in Groovy

```
 framework = "org.apache.felix.framework.Felix"
 lib = "file:jar/felix.jar"

 ////// Generic
 storage = new File("osgi-  storage").getAbsolutePath()
 storage.mkdirs()

 properties = [
 'org.osgi.framework.system.packages':"org.osgi.framework,\
    org.osgi.service.packageadmin, \
    org.osgi.service.startlevel,\
    javax.sql",
 'org.osgi.framework.storage' : storage.absolutePath,
'org.osgi.service.http.port'  : '8080'
 ]

 this.class.classLoader.rootLoader.addURL( new URL(lib) )

 clazz = Class.forName(framework)
 systemBundle = constructor.newInstance()
 systemBundle.init(properties)

 ctx     = systemBundle.bundleContext
 servlet = ctx.installBundle("http://www.osgi.org/repository/servlet.jar")
 osgi    = ctx.installBundle("http://www.osgi.org/repository/osgi.jar")
 webrpc  = ctx.installBundle("http://www.aqute.biz/uploads/Code/aQute.webrpc.jar")
 suduko  = ctx.installBundle("http://www.aqute.biz/uploads/Code/aQute.sudoku.jar")
 http    =
ctx.installBundle("http://www.knopflerfish.org/repo/jars/http/http_all-2.1.0.jar")

 http.start()
 webrpc.start()
 suduko.start()

 systemBundle.start()
```

The shell language, outlined in the following sections, can also be used in the same way. However, this is combined in a launcher program that gets the framework library and system bundle class from the command line paramaters:

```
 org.osgi.framework.system.packages ="
       org.osgi.framework,
       org.osgi.service.packageadmin,
       org.osgi.service.startlevel,
       javax.sql"
 org.osgi.framework.storage = $user.home/osgi
 org.osgi.service.http.port = 8080

 start
 ctx = bundleContext
 addCommand ctx $ctx
 servlet = installBundle http://www.osgi.org/repository/servlet.jar
 osgi    = installBundle http://www.osgi.org/repository/osgi.jar
 webrpc  = installBundle http://www.aqute.biz/uploads/Code/aQute.webrpc.jar
 suduko  = installBundle http://www.aqute.biz/uploads/Code/aQute.sudoku.jar
 http    = installBundle
http://www.knopflerfish.org/repo/jars/http/http_all-2.1.0.jar
```

This design has the tremendous advantage that each organization can use its own script language. Due to the abstraction of the system bundle, it is trivial to create a launcher that can be combined with any compliant framework. It is therefore not deemed wise to standardize the script syntax for the launch process, there are already a sufficient number around.

However, it is advantageous to take advantage of the command provider interface. This model is described later, but it is based on services. Using the script language approach as defined here, it is quite easy to search the service registry for new commands. The groovy meta model or the undefined command catch functions can make this quite transparent.

## 4.5 Shell Design

The drivers of this design have been:

- Core Engine Implementable in < 30k
- Very easy to add new commands
- Leverage existing mechanisms

The basic idea of the design is that there are three parts. The bundle that interacts directly with the user. This bundle handles the IO streams and parses out one or more lines of text, called the “program”. This program creates a Command Session from the selected Command service. This IO processor then gets a command from the input and gives it to a command session execute method. The command session parses the program, and executes it. The session then returns an Object result.

The command is executed synchronously. The shell will execute all *commands* in the program. These commands are implemented by services. A service can be registered with list of COMMAND_FUNCTION properties. These properties list the commands (potentially wildcarded) that a service can provide to the shell. These functions do not require a specific prototype, the shell matches the parameters to the function using parameter *coercion*. The type information available in the reflection API is used to convert the strings in the input to specific types.

Each command can print to System.out and it can retrieve information from the user (or previous command in the pipe) with System.in. Each command can also return an object. It is possible to retrieve the last result in the pipe through the future.

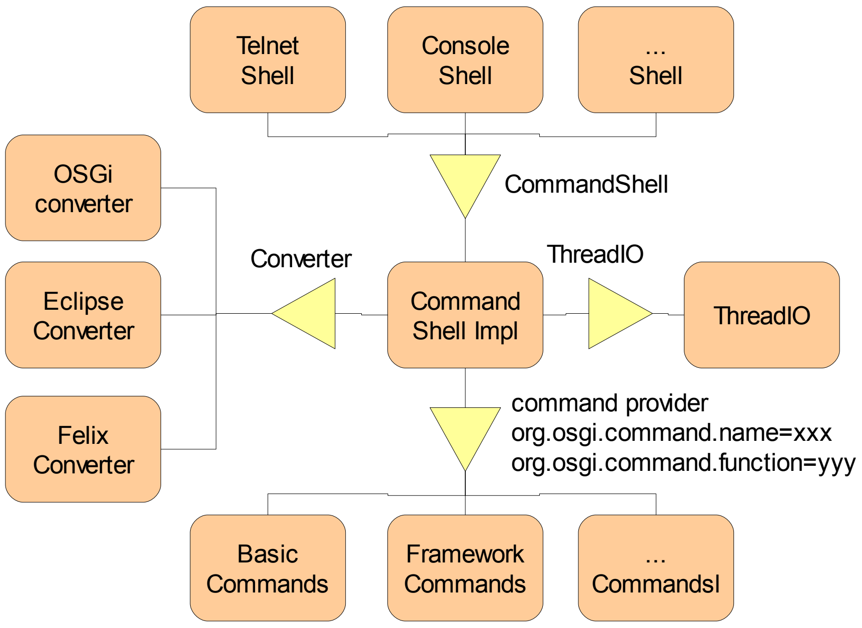
Therefore, the Command Shell service consists of three distinct parts:

- Command Processor service - This service is used by bundles that can connect the shell to an outside interface like: Telnet, Console, Web, SSH, etc. These bundles get a Command Shell service and use this service to execute their commands.

- Command Provider service - Command implementations can register this service to provide commands. Commands are methods on the service. The names (and help) of the methods can be listed through properties. There is no actual Command Provider interface because a service property allows any service to provide commands.

- Converter Service – Provide facilities to convert from strings to specific types and from specific types to strings.

- ThreadIO Service - Commands can use System.in, System.out, and System.err to interact with the user. However, this requires that different commands are separated in their output. This RFC therefore defines a service (likely a Framework service) that can multiplex the System IO streams.



## 4.6 Command Service

The command service consists of the following interfaces

- CommandProcessor – The engine is running the scripts. It has no UI of its own. A UI (telnet, web, console, etc, is expected to create a session from this engine. The Command Processor service is registered by the implementer of the script processor.
- CommandSession – The command session represents the link between a UI processor (telnet) and the command processor. It maintains a set of variables that can be set by the UI processor as well as from the script. Commands should maintain any state in the session. The session is also associated with a keyboard stream as well as a console stream. This allows commands to directly talk to the user, regardless if they are piped or not.
- Converter – A converter service is registered with a list of classes at the osgi.converter.classes properties. A converter can convert an object to a CharSequence object and it can convert an object of an arbitrary class (though likely a CharSequence) to an instance of one of the listed classes. Converters are heavily used to minimize the command code. The Command Processor will attempt to coerce parameters and results in the required instances using converters.
- Function – A function is an executable piece of code. Commands providers can add Function objects to their arguments and execute them. This allows commands that implement iteration blocks, if statements, etc.

The following code executes a small program, assuming it is injected with a CommandProcessor called cp:

```
ByteArrayOutputStream bout = new ByteArrayOutputStream();
CommandSession session = cp.createSession(System.in,bout,System.out);
Object result = session.execute("bundles|grep aQute");
String s  = new String(bout.toByteArray());
```

## 4.7 Thread IO Service

The Thread IO service is a framework service that guards the singletons of System.out, System.in, and System.err. The interface is quite simple, it consists of two methods:

- setStream(InputStream,PrintStream,PrintStream) – Associate the given streams with the current thread. Any output on the current thread using any of the System Print Streams will in effect be redirected to the approriate system stream. Input will come from the given input stream. This method can be repeated multiple times for a thread. That is, an implementation must stack the streams per thread. Streams may be null, in that case they refer to the last set stream or the default if no streams are set.
- close() - Cancel (or pop) the streams from the thread local stack. If no more streams are available, use the value of the original System streams.

Usage of the Thread IO service is very straightforward but care must be taken that exceptions do not leave streams on the stack. For example, the following code grabs the output:

```
String grab(ThreadIO threadio ) {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  PrintStream pout = new PrintStream(out);
  threadio.setStreams(null,pout,pout);
  try {
      System.out.println("Starting ...");
```

```
        doWhatever();
        System.out.println("... Done");
    } catch(Throwable t ) {
        t.printStackTrace();
    }
    finally {
      threadio.close();
    }
    pout.flush();
    return new String( out.toByteArray() );
  }
```

Additional issues:

- Implementations of Thread IO must use weak references to the stream objects and no longer use them when they  hold the only reference.
- If a framework is embedded then the threadio service must attempt to reuse the threadio implementation of the parent framework.

## 4.8 Shell syntax: TSL

The syntax of the shell should be simple to implement because the framework must provide a parser for this syntax. On the other hand, a more powerful syntax simplifies the implementation of the commands. For example, when Microsoft introduced a command line shell, it did not support piping. As a consequence , each command had to implement functionally to page the output. There are other examples like handling of variables, executing subcommands, etc.

The shell syntax must also be easy to use by a user. That is, a minimum number of parentheses, semicolons, etc. Some compatibility with the Unix shells like bash is desired to for users to not have to learn completely new concepts. Then again, the current popular shells have a convoluted syntax because they added more and more features over time.

An OSGi shell syntax can rely 100% on the fact that there is a Java VM. As shown in the launching section, this makes it easy to control the framework and implement a shell with Java. However, a shell implies that the users directly types the commands as they go. The requirements for a shell are therefore different than the requirements for a programming language. However, in contrast with a shell like bash that must be totally text based, it seems a waste not to tie the shell language closely to the Java object model. Though there are many script languages, there seems to be no shell language for Java that provide such a syntax. Jacl comes close but has the disadvantage that it brings its own function library derived from tcl. Beanshell comes close from the other side but has a syntax that is virtually the same as Java, which contains a lot of cruft characters. A new syntax can reuse the concepts of tcl but tie the language close to the Java language. I.e., no need for separate function libraries.

### 4.8.1 Introduction to TSL (Tiny Shell  Language).

The tsl language consists of a tokenizer that converts a command line to strings and then interprets those strings as method calls on Java objects where the arguments are converted to the requested type. Some syntactic sugar is added to minimize typing. The language supports variables, which are real objects. Do not confuse tsl with a completely string based approach. The type information available in the Java VM is used to infer types and convert to proper objects as much as possible.

Some examples:

```
$ echo Hello World
```

```
  Hello World
```

The first token is the command name because it is a simple string, and is a *command*. A command represents both an *instance* and the *method* to be called on that instance. This is in first instance confusing. However, the way it works is quite simple. Objects implementing a command (that is, having a method with a command name) are registered as a variable with a structure. The same object can be registered under many different names. That is, if an object implements ls and cd, then it is registered as file:ls, and file:cd.

As can be seen in this exam,ple, the actual name of the command is a structured name consisting of a <scope> and a function name, separated by a ':'. I.e. in the earlier example the command '*:echo' is searched because the scope is not defined.

The commands are coming from the variable space of the session. It must be possible to register commands by creating variables of the proper name.

A command is represented by a Function object. In the hello world examples, the execute method is called on this object with two CharSequence parameters: [“Hello”,”World”]. Because the number of arguments of echo is variable, it is declared with an array of Object.

```
  public CharSequence echo( Object[] args ) {
    StringBuilder sb = new StringBuilder();
    for ( Object arg : args )
      sb.append(arg);
    return sb;
  }
```

Methods can print to System.out, but are normally expected to return an object. Returning an object allows the result to be used in other commands. Tsl will print out the object to standard out if it is not used as a value in another command, for example with piping. If a program contains multiple statements, only the last value is printed out. Converter services are used to print out the objects in a proper format.

### 4.8.2 Program Syntax

The basic structure of a *program* is a set of statements separated by a vertical bar.

```
  program ::= statements ( '|' statements ) *
```

The *statements* are executed in parallel, whereby the output of the earlier statement is the input of the next statements. These statements are executed in separate threads. Statements consist of a sequence of statement, separated by a semicolon.

```
  statements ::= statement ( ';' statement ) *
```

A statement is initially a set of tokens. This means that they must be tokenized in preparation for a command execution.

```
  statement ::= ( WS* token )+
```

A token is parsed out of the input. Important is the first character, this determines the type of the token. If the first character is a Java Identifier part (digit, alpha, underscore, etc), then the token is parsed until an unescaped WS is found or one of the special characters is found. with the following rules:

```
  token     ::= JIP (JIP | ^SPECIAL )*
            |  ^SPECIAL+
            | '<' <recursive> '>'
            | '{' <recursive> '}'
            | '[' <recursive> ']'
            | '(' <recursive> ')'
            | ''' [^'] '''
            | '"' [^"] '"'
            | '$' token
  WS        ::= <Character.isWhiteSpace>
  JIP       ::= <Character.isJavaIdentifierPart - $>
  SPECIAL   ::= [=|;<{[$,]
  OPERATORS ::= [!~`#$%^&*-:,/?@.]
```

The <recursive> indicates that the token should be parsed until a matching closing bracket is found, the parsing should take escaping and strings into account finding the match. I.e. <<<<>>>, matches, as does <'><'>, as does {'}}}}}}}}}'}. Though the content is expected to be a valid program, the tokenizer does not have to verify this, nor is this mandated. Implementing such a tokenizer is quite straightforward and requires very little code. This is the reason that the syntax does not use the program clause recursively but instead comments that it is a recursive match.

Character handling. Some characters can escaped from special processing with the backslash character. These characters will be interpreted as their normal value without any special meaning. Some other characters escaped will give special codes:

| | |
|---|---|
| `\\` | `Backslash (2dh###)` |
| `\t` | `Tab (whitespace) 09h)` |
| `\b` | `backspace (08h)` |
| `\f` | `Form Feed (0ch)` |
| `\n` | `New Line (0ah)` |
| `\r` | `Return (0dh)` |
| `\u9999` | `Unicode` |
| `\<lf \| cr>` | `Make newline a space` |
| `\` | `Escaped whitespace` |

```
### need some further work on escaping ...
```

Whitespace is defined by Java. Unless escaped or in a string, whitespace has no meaning.

The tokens that are parsed out of the input have a special meaning depending on their first character:

- < - The is less than sign indicates a direct execution block. Everything between this sign and the closing greater than sign must be a valid program. This is equivalent to a function call or the ` ..` operator in a unix shell. This operator was considered but unfortunately there is no difference between the opening and closing in unix, requiring complex escaping with recursive use. Everything between the opening and '<' and matching '>' belongs to the direct execution. That is, the direct execution can contain  be recursive to any reasonable depth.

- '{' - The closure character. The closure contains statements. It is defined until the  matching closing brace '}'. Closures are not interpreted by the shell but passed as is a Closure object the command. The first character will be the {. Closures can be nested to any reasonable depth. No unescaping is done. This is the responsibility of the receiver. The purpose of this method is to allow commands to use closures. If a command declares a Function argument, then tsl must convert a closure to a Function. Closures can refer to any parameters by using the variables $it (the first argument), $args (an array of all the arguments) and $0-$9, argument at position 0 through 9.

- '[' - The array character. An array is a whitespace separated sequence of *entry*. This is stored in a Collection or Map and can be coerced to arrays, and collection types. Arrays/Maps can be recursively defined.

- '"' and ''' - The quote characters define a string. Everything between the quote and its matching end quote is passed as a String. All characters will be unescaped when passed as an argument.

- '$' - The variable character. If the next character is a '{' then the code between  this and the matching '}' is a value, which can be a full program again. The value of this program in runtime is used to lookup a value in the System properties.

- '(' - The expression character. Everything between this character and the closing ')' defines a filter expression. This token can be coerced into a filter or a string for a function.  A filter can  use '(' and ')' recursively.

An array is a sequence of *token* separated by whitespace.

```
array      ::=  (WS* entry)*
entry      ::=   token  ( WS* '=' token  )*
```

### 4.8.3 Examples of Syntax usage

In the following examples it is assumed that there is an echo command which prints the output to System.out.

## 4.9 Standard IO Handling

The original shells in Apache Felix, Knopflerfish, and Equinox used special handling of input and output (if input was supported). A huge disadvantage of this method is that it requires all the command to live in a special context; making the commands hard to test. Reuse of existing code is also harder because it is likely that any io must be adapted.

This RFC therefore proposes the use standard Java input output with the System.in, System.out, and System.err streams. This means that any Java program using these streams will work. However, these streams are singletons and Java does not provide a general way to share these singletons between bundles. This would create conflicts if multiple shells were running on the system. Even if one shell runs this would be problematic because a pipeline uses different threads that need different IO streams.

This requires a service that can multiplex the IO streams based on the current thread. In practice this is almost trivial to do (a test class uses less than 50 lines, where most are actually whitespace) with Thread Local Variables. However, the key problem is to the synchronization between the different users because it requires replacing the existing System.in, out, and err with a special multiplexing class. Just like the URL services, these are singletons. We therefore need a (framework?) service that allows bundles to associate IO streams with a thread.  This is a very useful function in itself.

The setStreams method will associate the given IO streams with the current thread. Any code using System.out, System.in, or System.err will use the given streams instead of the standard streams. The close method will restore the previous configuration (the streams will be pushed on a stack). If the bundle that used this service is stopped, then the stack of streams will be removed.

The Shell service uses this mechanism to associate the streams from the shell drivers with the commands, as well as for the piping.

## 4.10 Command Provider Discovery

Command Provider discovery is based on the OSGi service model. Any service can be used as a command provider.

Dedicated command providers must register their service with two properties:

- osgi.command.scope - This property defines the name of the command provider. This name is not normally used because the function names are unique. However, if the function names are  no longer unique, then this scope can be used to disambiguate.

- osgi.command.function - The name of the function. This is an simple or an array property, so many names can be listed. This function name should match to a public method in the service object.

For example, the following code is a DS that provides a few utility commands:

```
  public class Tools {
    public void grep(String match) throws IOException {
      Pattern p = Pattern.compile(match);
      BufferedReader rdr = new BufferedReader(
         new InputStreamReader(System.in));
      String s = rdr.readLine();
      while (s != null) {
        if (p.matcher(s).find()) {
           System.out.println(s);
        }
        s = rdr.readLine();
      }
  }
  public void echo( Object[] args) {
    StringBuffer sb = new StringBuffer();
    for ( Object arg : args )
      sb.append(arg);
  }
}
```

This command provider can support two commands: echo and grep. The DS scheme for this command would look like:

```
<component name="com.acme.Tools">
  <implementation class="com.acme.Tools"/>
    <property name="command.scope" value="acme.tools"/>
    <property name="command.function" value="
      grep
      echo"/>
    <service>
      <provide interface="com.acme.Tools"/>
    </service>
</component>
```

The properties provide sufficient information for the Command Shell to find the providers. Note that it is not necessary register the service as a Command Provider, the properties suffice. This makes it possible to register these properties on an existing service. For example, the Configuration Admin could just register the following properties:

```
  osgi.command.scope = 'cm'
  osgi.command.function = { 'createFactoryConfiguration',
     'getConfiguration', 'listConfigurations'}
```

This will enable shell scripts like:

```
  cfg = configuration com.acme.pid
  $cfg update [port=23 host=www.acme.com]
```

Or, for the Log Service

```
command.scope = 'log'
command.function = 'log'

log 2 "hello world"
```

## 4.11 Other Commands

Any other commands can be added to the shell by storing them in the session variables. Command names are scoped like <scope>:<function>. The value of this variable can be a plain object, or it can be an instance of Function. If it is an instance of Function, it can be directly executed. Else, the method with the function name is called upon it.

For example, the following code registers a function for each public method:

```
void addCommand(CommandSession session, String scope, Object target ) {
  Method methods[] = target.getClass().getDeclaredMethods();
  for ( Method m : methods ) {
    if ( Modifiers.isPublic(m.getModifiers()))
      session.put( scope +":" +m.getName(), target );
  }
}
```

Tsl also has *closures*. Closures implement the Function interface. The follow code will add a command written in tsl:

```
$ my:echo = { echo xx $args xx }
Closure ...
$ my:echo Hello World
xxHelloWorldxx
```

## 4.12 Piping

Piping seems to introduce a significant complexity in the command processing. However, it turns out that it can be implemented with very little code that easily outweighs the advantages if the increased simplicity of the commands. The key example is of course the "less" or "more" command. Many of the other commands can generate output that is too much to fit the screen. Using piping, the output can be paged through a centralized command. Functions like grep, uniq, etc. are all impossible without piping.

Normal unix shells have io redirection. It was chosen to not implement this, but instead use commands. There ar two commands that can redirect io:

```
cat <file>+          concatenate files and pipe to output
tac [-f <file>]      receive input and store it in file or return object
```

That is, to get the output of bundles as a string in a variable:

```
output = <bundles | tac>
```

## 4.13 Command Calling

When the program is tokenized it basically consists of set of statements. A statement consists of a set of tokens. These tokens are parsed into objects. I.e. a reference to a variable is looked up, a <> is executed recursively, a {} is translated to a Closure, an array into a List or Map. The next step depends on the list of values.

```
<string> '='                remove variable <string>
<string> '=' <value> +      execute values as statements, set result as variable
<string>                    if cmd exists, call <string>, else no such command
```

```
<string> <value>+             call cmd <string> with arguments
<object> <value> <value>*     send message <value> to <object> with remaining args
```

### 4.13.1 Remove Variable

The form <name> '=' is used to remove a variable from the local scope. There must be no token behind the '=', not even an empty token, like for example ''

```
java.lang.vm =
```

### 4.13.2 Assignment

An assignment has the form:

```
<property name> '=' statement
```

The statements part must be executed as if it was entered on an empty line, the result of this execution is stored in the variable. For example:

```
jre-1.6 = javax.xml.parsers, javax....
org.osgi.framework.systempackages =  ${jre-${java.specification.version}}
```

### 4.13.3 Single Value

This is an interesting case. If the single value is a string, then it can be a command name, or an object that needs needs to be returned. If it is not a CharSequence, it is assumed to be an object and it is the value of the statement. If it is a CharSequence, a command with the given name is searched. If it is found, it is executed according to the normal rules. If not, it is assumed that the string value suffices as result.

```
 abcefg                             // result no such command
 $shell                             // returns the content of the variable
 <abcef>                            // returns no such command
```

### 4.13.4 Call Cmd

When the first argument is a string, it is assumed to be a command name. This command is looked up in the list of available Command Providers. If this is not found, an error is raised unless there are no parameters . In that case the name is returned as the result.

If a Command Provider is found, then this object is the target for a message send, the name of the message being the command name. See discovery. The arguments are matched as discussed in the “Argument Coercion” section.

```
echo 1
```

### 4.13.5 Message Send

If the first argument is not a string but the second is, then a message send is assumed. The method with the given name is matched to the remaining arguments.

```
shell = install http://www.acme.com/b/shell.jar // sets a bundle object
$shell start                                    // starts the bundle
```

If there are no arguments and no method matches, then the name should be treated as a property name. The method name should be adjusted to the beans get property design pattern (i.e. xyz becomes getXyz). If this method also does not exist, a field with the given name should be tried. I.e. the following command should return the requested level as an integer:

```
log LOG_WARNING
```

## 4.14 Argument Coercion

In the end, a statement consists of an assignment or the call of a Java function. When a Java function is called, it is necessary to match the arguments to the correct method. This requires that arguments are coerced in their correct type.

First finding the proper command in the service. If the cmd is a reserved word in java (static, final, new, etc.) then the command name must be prefixed with a '_' because otherwise the command could not be implemented in Java. That is, if the command is “new”, the method name should be “_new”.

The comparison with the method name must be done case insensitive. I.e. installBundle is the same as installbundle.

The bean syntax must be supported. That is, a command like bundles must find the method getBundles and setBundles and isBundles. See the beans design patterns for proper converting a name to a getter.

Then, the method must be found. The shell should fetch all the declared methods of the service and try to match the given parameters to all the methods with the command name. This requires matching the arguments given in the program to the arguments of the method. Matching of the methods is done with the following priority:

1. The first declared method where all arguments can be properly coerced

2. If too few arguments are specified, pad with null the method with the maximum matching arguments. Done

3. If too many arguments are specified, the find the first method which has an array at the end and where the remaining arguments can be coerced into to the array type. Done.

4. An instance “main(Object[])” method

5. The static main(String[]) method

If none is found, a NoSuchMethodException must be thrown.

If the first type of a command is a CommandSession, then the Command Shell must insert the current session in this parameter. This is a way for a command to receive the shell session itself. This can be used to recursively execute commands, to get access to the keyboard or console stream, or to get and put variables. The keyboard stream is for example necessary to do a more command.

Arguments are no strings, they are proper objects. Variables can refer to objects, arrays are objects, and also the result of a direct command (<>) can result in a proper object. Matching these objects to a method is non-trivial. In the following section r is the receiving type (defined in the method) and g is the given type. The priority for coercion is:

1. r is assignable from g, use g

2. r is an array and g is a Collection, convert g to an array r and coerce its members recursively

3. Iterate over all converters that can handle the requested type as listed in the properties, until one returns a non-null value. Order is service.ranking and the service.id.

4. r is not primitive and has String constructor, convert g to string and use constructor to convert g to r.

5. r is a primitive and g is the matching class, convert g to primitive value and use it
6. fail

## 4.15 Converters

In most shells, the formatting of the input is the majority of work. The tsl has attempted to minimize this work with the Converter services.

A Converter service registers itself with a list of class names that it can convert or format. Tsl uses these services to print  returned objects or to coerce arguments for method calls.

The service property is osgi.converter.classes. Its value is a single string or an array of strings, reflecting the classes this converter can convert or print. For conversion, inheritance is not taken into account. For printing, tsl must start with the implementation class, then its superclass, recursively. If no match is found, it should try to find all implemented interfaces in the class and its ancestors. For each converter it should combine the output.

### hmmm, not sure I like this.

When tsl needs to convert an object to a class or print an object of a specific class, it will call the registered Converter objects in the following order:

- filtered by matching class
- sorted by service.ranking, service.id

A Converter service implements 2 methods:

- Object convert(Class,Object) – Convert object to the given class. Return null if this can not be done.
- CharSequence format(Object,int) – Convert an object to a Char Sequence using the int parameter as a hint. This hint can be INSPECT, LINE, or PART. For an INSPECT, the output can be a multiline columnar output of any reasonable level. A LINE format must make the object look good in a table when different objects of the same type are printed below each other. It is allowed to use multiple line outputs as long as the format works well in a table. A PART format is used to identify the object. E.g. a name or identifier. The PART format should be usable in the convert method when a CharSequence is the object to be converted. INSPECT, LINE, and PART are ordered. That is, when printing an INSPECT, the next level should be to format an object with LINE, etc.

The following code shows a simple converter for Bundles that only recognizes the bundle id. (A real one should also look for symbolic name and version, or for the location, or maybe even a filter). The printing should be much better in aligning columns.

```
import org.osgi.framework.*;
import org.osgi.service.command.*;

public class BundleConverter implements Converter {
    BundleContext context;

    BundleConverter(BundleContext context) {
        this.context = context;
    }

    public Object convert(Class type, Object source) {
        if (type != Bundle.class)
```

```
            return null;

        if (source instanceof Number) {
            source = source.toString();
        }

        if (source instanceof CharSequence) {
            long id = Long.parseLong(source.toString());
            return context.getBundle(id);
        }
        return null;
    }

    public CharSequence format(Object o, int level, Converter escape) {
        if (!(o instanceof Bundle))
            return null;

        Bundle b = (Bundle) o;
        StringBuffer sb = new StringBuffer();
        switch (level) {
        case INSPECT:
            cols(sb, "Symbolic Name", b.getSymbolicName());
            cols(sb, "Version", b.getHeaders().get("Bundle-Version"));
            cols(sb, "State", b.getState());
            cols(sb, "Registered Services", escape.format(b
                    .getRegisteredServices(), level + 1, escape));
            // ...
            break;

        case PART:
            sb.append(b.getSymbolicName()).append(";").append(
                    b.getHeaders().get("Bundle-Version"));
            break;

        case LINE:
            sb.append(" ").append(b.getState()).append(" ").append(
                    b.getLocation());
            break;
        }
        return sb;
    }

    void cols(StringBuffer sb, String label, Object value) {
        sb.append(label);
        for (int i = label.length(); i < 24; i++)
            sb.append(' ');
        sb.append(value).append('\n');
    }
}
```

## 4.16 Printing or Not

In principle, tsl must only print the object when it would otherwise gets lost. It will therefore only print the object when a command is piped because in that case there is nobody to use the resulting object. In all other cases, the object is kept.

## 4.17 TSL In OSGi

If the use of tsl is OSGi related then it will have registered commands for all the public methods on the BundleContext, StartLevelService, PackageAdmin, and PermissionAdmin (if present). The actual bundle context in use is from the IO Processor. The scope of the bundle context commands must be osgi. The actual Bundle Context must be from the Bundle that got the Command Processor. This is usually the IO processor, e.g., the telnet or console program.

Additionally, the following services should be supported:

- *more* – Page the input
- *grep <regex>* – Search in the input and only transfer to output the input that matches the <regex>.
- *each <Iteratable> <closure>* - Iterate over the iterable and call the closure for each element. The first argument ($it) is the iterable element.
- *echo <value>* * - Print the value to the System.out without any spaces in between.
- *quit* – Quits the shell
- *exit* – Exits the framework

This makes any public method available:

```
$ bundles | grep aQute
0003 ACT biz.aQute.bnd               file:/Ws/aQute/aQute.bnd/bnd.jar
0023 ACT biz.aQute.fileinstall       file:/Ws/aQute/aQute.fileinstall/fileinstall.jar
```

The bundles command is found as osgi:bundles. This command has the current Bundle Context (defined by the env again) as the implicit receiver and the name bundles as function. There is no function bundles on Bundle Context, but in the spirit of the beans, tsl must also look at no arg get methods. I.e. the method name is getBundles but for brevity, bundles must be matched as well. This method returns null or a Bundle[].

Because this command is input from the user, and piped, tsl will print it to the pipe. The grep function looks like:

```
public CharSequence grep(String match) throws IOException {
  Pattern p = Pattern.compile(match);
  BufferedReader rdr = new BufferedReader(
    new InputStreamReader(System.in));
  List<String> list = new ArrayList<String>();
  StringBuilder sb = new StringBuilder();
  String s = rdr.readLine();
  while (s != null) {
    if (p.matcher(s).find()) {
      list.add(s);
    s = rdr.readLine();
  }
  return list;
}
```

## 4.18 Services and their Commands

Implementations of services are recommended to provide commands for their service, this is quite straightforward and described in a later section. For example, assume that the implementation of te Configuration Admin has registered its method as commands. I.e. all its public methods are available.

```
$ my.pid = configuration my.pid; $my.pid update [port=5012 host=www.aQute.biz ]
```

The configuration command is executed against the Configuration Admin service. This returns a Configuration object. In this case, it is stored in the my.pid variable. In the next statement, we call the update method on the Configuration object and set a dictionary.

Ok, one more example, based on the fact that the Configuration Admin is available as commands:

```
 $ listConfigurations (service.pid=com.acme.*)| grep port
```

## 4.19 Help

tbd

# 5 Javadoc

**Overview** **Package** **Class** **Use** **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

## 5.1 org.osgi.framework.launch
## Interface SystemBundle

**All Superinterfaces:**
Bundle

`public interface SystemBundleextends Bundle`

This interface should be implemented by framework implementations when their main object is created. It allows a configurator to set the properties and launch the framework. TODO The javadoc in this class need a good scrub before release.

**Version:**
$Revision: 5214 $

| Field Summary | |
|---|---|
| `static java.lang.String` | **EXECPERMISSION**<br>The command to give a file executable permission. |

| | |
|---|---|
| `static java.lang.String` | **LIBRARIES** A list of paths (separated by path separator) that point to additional directories to search for platform specific libraries |
| `static java.lang.String` | **ROOT_CERTIFICATES** Points to a directory with certificates. |
| `static java.lang.String` | **SECURITY** The name of a Security Manager class with public empty constructor. |
| `static java.lang.String` | **STORAGE** A valid file path in the file system to a directory that exists. |
| `static java.lang.String` | **WINDOWSYSTEM** Set by the configurator but the framework should provide a reasonable default. |

| Fields inherited from interface org.osgi.framework.Bundle |
|---|
| `ACTIVE, INSTALLED, RESOLVED, START_ACTIVATION_POLICY, START_TRANSIENT, STARTING, STOP_TRANSIENT, STOPPING, UNINSTALLED` |

## Method Summary

| | |
|---|---|
| `void` | **init**`(java.util.Properties configuration)` Configure this framework with the given properties. |
| `void` | **waitForStop**`(long timeout)` Wait until the framework is completely finished. |

| Methods inherited from interface org.osgi.framework.Bundle |
|---|
| `findEntries, getBundleContext, getBundleId, getEntry, getEntryPaths, getHeaders, getHeaders, getLastModified, getLocation, getRegisteredServices, getResource, getResources, getServicesInUse, getState, getSymbolicName, hasPermission, loadClass, start, start, stop, stop, uninstall, update, update` |

## Field Detail

### 5.1.1 SECURITY

`static final java.lang.String` **`SECURITY`**

The name of a Security Manager class with public empty constructor. A valid value is also true, this means that the framework should instantiate its own security manager. If not set, security could be defined by a parent framework or there is no security. This can be detected by looking if there is a security manager set

**See Also:**
Constant Field Values

### 5.1.2 STORAGE

```
static final java.lang.String STORAGE
```

A valid file path in the file system to a directory that exists. The framework is free to use this directory as it sees fit. This area can not be shared with anything else. If this property is not set, the framework should use a file area from the parent bundle. If it is not embedded, it must use a reasonable platform default.

**See Also:**
Constant Field Values

### 5.1.3 LIBRARIES

```
static final java.lang.String LIBRARIES
```

A list of paths (separated by path separator) that point to additional directories to search for platform specific libraries

**See Also:**
Constant Field Values

### 5.1.4 EXECPERMISSION

```
static final java.lang.String EXECPERMISSION
```

The command to give a file executable permission. This is necessary in some environments for running shared libraries.

**See Also:**
Constant Field Values

### 5.1.5 ROOT_CERTIFICATES

```
static final java.lang.String ROOT_CERTIFICATES
```

Points to a directory with certificates. ###??? Keystore? Certificate format?

**See Also:**
Constant Field Values

### 5.1.6 WINDOWSYSTEM

```
static final java.lang.String WINDOWSYSTEM
```

Set by the configurator but the framework should provide a reasonable default.

**See Also:**
Constant Field Values

## Method Detail

### 5.1.7 init

```
void init(java.util.Propertiesconfiguration)
```

Configure this framework with the given properties. These properties can contain framework specific properties or of the general kind defined in the specification or in this interface.

**Parameters:**
`configuration` - The properties. This properties can be backed by another properties, it can there not be assumed that it contains all keys. Use it only through the getProperty methods. This parameter may be null.

### 5.1.8 waitForStop

```
void waitForStop(longtimeout)
                  throws java.lang.InterruptedException
```

Wait until the framework is completely finished. This method will return if the framework is stopped and has cleaned up all the framework resources.

**Parameters:**
`timeout` - Maximum number of milliseconds to wait until the framework is finished. Specifying a zero will wait indefinitely.
**Throws:**
`java.lang.InterruptedException` - When the wait was interrupted

Overview Package **Class** Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package **Class** Use Tree Deprecated Index Help
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

## 5.2 org.osgi.service.command
## Interface CommandProcessor

```
public interface CommandProcessor
```

A Command Processor is a service that is registered by a script engine that can execute commands. A Command Processor is a factory for Command Session objects. The Command Session maintains execution state and holds the console and keyboard streams. A Command Processor must track any services that are registered with the COMMAND_SCOPE and COMMAND_FUNCTION properties. The functions listed in the COMMAND_FUNCTION property must be made available as functions in the script language. TODO The javadoc in this class need a good scrub before release.

**Version:**
$Revision: 5214 $

## Field Summary

| | |
|---|---|
| static java.lang.String | **COMMAND_FUNCTION** A String, array, or list of method names that may be called for this command provider. |
| static java.lang.String | **COMMAND_SCOPE** The scope of commands provided by this service. |

## Method Summary

| | |
|---|---|
| CommandSession | **createSession**(java.io.InputStream in, java.io.PrintStream out, java.io.PrintStream err) Create a new command session associated with IO streams. |

## Field Detail

### 5.2.1 COMMAND_SCOPE

```
static final java.lang.String COMMAND_SCOPE
```

The scope of commands provided by this service. This name can be used to distinguish between different command providers with the same function names.

**See Also:**
Constant Field Values

### 5.2.2 COMMAND_FUNCTION

```
static final java.lang.String COMMAND_FUNCTION
```

A String, array, or list of method names that may be called for this command provider. A name may end with a *, this will then be calculated from all declared public methods in this service. Help information for the command may be supplied with a space as separation.

**See Also:**
Constant Field Values

## Method Detail

### 5.2.3 createSession

```
CommandSession createSession(java.io.InputStreamin,
                             java.io.PrintStreamout,
                             java.io.PrintStreamerr)
```

Create a new command session associated with IO streams. The session is bound to the life cycle of the bundle getting this service. The session will be automatically closed when this bundle is stopped or the service is returned. The shell will provide any available commands to this session and can set additional variables.

**Parameters:**
`in` - The value used for System.in
`out` - The stream used for System.out
`err` - The stream used for System.err

**Returns:**
A new session.

## 5.3 org.osgi.service.command
## Interface CommandSession

`public interface CommandSession`

A Command Session holds the executable state of a script engine as well as the keyboard and console streams. A Command Session is not thread safe and should not be used from different threads at the same time. TODO The javadoc in this class need a good scrub before release.

**Version:**
$Revision: 5214 $

## Method Summary

| | |
|---|---|
| `void` | `close()`<br>Close this command session. |
| `java.lang.Object` | `convert(java.lang.Class type, java.lang.Object instance)`<br>Convert an object to another type. |
| `java.lang.Object` | `execute(java.lang.CharSequence commandline)`<br>Execute a program in this session. |
| `java.lang.Object` | `execute(java.lang.CharSequence commandline, java.io.InputStream in, java.io.PrintStream out, java.io.PrintStream err)`<br>Execute a program in this session but override the different streams for this call only. |
| `java.lang.CharSequence` | `format(java.lang.Object target, int level)`<br>Convert an object to string form (CharSequence). |
| `java.lang.Object` | `get(java.lang.String name)`<br>Get the value of a variable. |
| `java.io.PrintStream` | `getConsole()`<br>Return the PrintStream for the console. |
| `java.io.InputStream` | `getKeyboard()`<br>Return the input stream that is the first of the pipeline. |
| `void` | `put(java.lang.String name, java.lang.Object value)`<br>Set the value of a variable. |

## Method Detail

### 5.3.1 execute

```
java.lang.Object execute(java.lang.CharSequencecommandline)
                         throws java.lang.Exception
```

Execute a program in this session.

**Parameters:**
`commandline` - ###
**Returns:**

the result of the execution

**Throws:**

`java.lang.Exception` - ###

---

### 5.3.2 execute

```
java.lang.Object execute(java.lang.CharSequencecommandline,
                         java.io.InputStreamin,
                         java.io.PrintStreamout,
                         java.io.PrintStreamerr)
                         throws java.lang.Exception
```

Execute a program in this session but override the different streams for this call only.

**Parameters:**

`commandline` -

`in` - ###

`out` - ###

`err` - ###

**Returns:**

the result of the execution

**Throws:**

`java.lang.Exception` - ###

---

### 5.3.3 close

```
void close()
```

Close this command session. After the session is closed, it will throw IllegalStateException when it is used.

---

### 5.3.4 getKeyboard

```
java.io.InputStream getKeyboard()
```

Return the input stream that is the first of the pipeline. This stream is sometimes necessary to communicate directly to the end user. For example, a "less" or "more" command needs direct input from the keyboard to control the paging.

**Returns:**

InpuStream used closest to the user or null if input is from a file.

---

### 5.3.5 getConsole

```
java.io.PrintStream getConsole()
```

Return the PrintStream for the console. This must always be the stream "closest" to the user. This stream can be used to post messages that bypass the piping. If the output is piped to a file, then the object returned must be null.

**Returns:**
###

### 5.3.6 get

```
java.lang.Object get(java.lang.Stringname)
```

Get the value of a variable.

**Parameters:**
`name` - ###
**Returns:**
###

### 5.3.7 put

```
void put(java.lang.Stringname,
         java.lang.Objectvalue)
```

Set the value of a variable.

**Parameters:**
`name` - Name of the variable.
`value` - Value of the variable

### 5.3.8 format

```
java.lang.CharSequence format(java.lang.Objecttarget,
                              intlevel)
```

Convert an object to string form (CharSequence). The level is defined in the Converter interface, it can be one of INSPECT, LINE, PART. This function always returns a non null value. As a last resort, toString is called on the Object.

**Parameters:**
`target` -
`level` -
**Returns:**

###

### 5.3.9 convert

```
java.lang.Object convert(java.lang.Classtype,
                         java.lang.Objectinstance)
```

Convert an object to another type.

**Parameters:**
`type` - ###
`instance` - ###
**Returns:**
###

## 5.4 org.osgi.service.command
## Interface Converter

```
public interface Converter
```

A converter is a service that can help create specific object types from a string, and vice versa. The shell is capable of coercing arguments to the their proper type. However, sometimes commands require extra help to do this conversion. This service can implement a converter for a number of types. The command shell will rank these services in order of service.ranking and will then call them until one of the converters succeeds. TODO The javadoc in this class need a good scrub before release.

**Version:**
$Revision: 5214 $

## Field Summary

| | |
|---|---|
| `static java.lang.String` | **CONVERTER_CLASSES**<br>This property is a string, or array of strings, and defines the classes or interfaces that this converter recognizes. |
| `static int` | **INSPECT**<br>Print the object in detail. |
| `static int` | **LINE**<br>Print the object as a row in a table. |
| `static int` | **PART**<br>Print the value in a small format so that it is identifiable. |

## Method Summary

| | |
|---|---|
| `java.lang.Object` | **`convert`**`(java.lang.Class desiredType, java.lang.Object in)`<br>Convert an object to the desired type. |
| `java.lang.CharSequence` | **`format`**`(java.lang.Object target, int level, Converter escape)`<br>Convert an object to a CharSequence object in the requested format. |

## Field Detail

### 5.4.1 CONVERTER_CLASSES

`static final java.lang.String` **`CONVERTER_CLASSES`**

This property is a string, or array of strings, and defines the classes or interfaces that this converter recognizes. Recognized classes can be converted from a string to a class and they can be printed in 3 different modes.

**See Also:**
Constant Field Values

### 5.4.2 INSPECT

`static final int` **`INSPECT`**

Print the object in detail. This can contain multiple lines.

**See Also:**
Constant Field Values

### 5.4.3 LINE

`static final int` **`LINE`**

Print the object as a row in a table. The columns should align for multiple objects printed beneath each other. The print may run over multiple lines but must not end in a CR.

**See Also:**
Constant Field Values

---

### 5.4.4 PART

```
static final int PART
```

Print the value in a small format so that it is identifiable. This printed format must be recognizable by the conversion method.

**See Also:**
Constant Field Values

## Method Detail

### 5.4.5 convert

```
java.lang.Object convert(java.lang.ClassdesiredType,
                         java.lang.Objectin)
                         throws java.lang.Exception
```

Convert an object to the desired type. Return null if the conversion can not be done. Otherwise return and object that extends the desired type or implements it.

**Parameters:**
`desiredType` - The type that the returned object can be assigned to
`in` - The object that must be converted

**Returns:**
An object that can be assigned to the desired type or null.

**Throws:**
`java.lang.Exception`

---

### 5.4.6 format

```
java.lang.CharSequence format(java.lang.Objecttarget,
                              intlevel,
                              Converterescape)
                              throws java.lang.Exception
```

Convert an object to a CharSequence object in the requested format. The format can be INSPECT, LINE, or PART. Other values must throw IllegalArgumentException.

**Parameters:**
`target` - The object to be converted to a String
`level` - One of INSPECT, LINE, or PART.

`escape` - Use this object to format sub ordinate objects.

**Returns:**

A printed object of potentially multiple lines

**Throws:**

`java.lang.Exception`

**Package** **Class** **Use** **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

**Package** **Class** **Use** **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

## 5.5 org.osgi.service.command
## Interface Function

```
public interface Function
```

A Function is a a block of code that can be executed with a set of arguments, it returns the result object of executing the script. TODO The javadoc in this class need a good scrub before release.

**Version:**

$Revision: 5214 $

## Method Summary

| | |
|---|---|
| `java.lang.Object` | `execute(CommandSession session, java.util.List arguments)` Execute this function and return the result. |

## Method Detail

### 5.5.1 execute

```
java.lang.Object execute(CommandSessionsession,
                         java.util.Listarguments)
                         throws java.lang.Exception
```

Execute this function and return the result.

**Parameters:**
`session` - ###
`arguments` - ###

**Returns:**
the result from the execution.

**Throws:**
`java.lang.Exception` - if anything goes terribly wrong

## 5.6 org.osgi.service.threadio
## Interface ThreadIO

```
public interface ThreadIO
```

Enable multiplexing of the standard IO streams for input, output, and error. This service guards the central resource of IO streams. The standard streams are singletons. This service replaces the singletons with special versions that can find a unique stream for each thread. If no stream is associated with a thread, it will use the standard input/output that was originally set. TODO The javadoc in this class need a good scrub before release.

**Version:**
$Revision: 5214 $

## Method Summary

| | |
|---|---|
| `void` | `close()`<br>Cancel the streams associated with the current thread. |
| `void` | `setStreams(java.io.InputStream in, java.io.PrintStream out, java.io.PrintStream err)`<br>Associate this streams with the current thread. |

## Method Detail

### 5.6.1 setStreams

```
void setStreams(java.io.InputStreamin,
                java.io.PrintStreamout,
                java.io.PrintStreamerr)
```

Associate this streams with the current thread. Ensure that when output is performed on System.in, System.out, System.err it will happen on the given streams. The streams will automatically be canceled when the bundle that has gotten this service is stopped or returns this service.

**Parameters:**

`in` - InputStream to use for the current thread when System.in is used
`out` - PrintStream to use for the current thread when System.out is used
`err` - PrintStream to use for the current thread when System.err is used

### 5.6.2 close

```
void close()
```

Cancel the streams associated with the current thread. This method will not do anything when no streams are associated.

# 6 Alternatives

### Maybe we should move the syntax to this section. Because we have standard launching the necessity of a standard syntax has become less. Hmm.

## 6.1 Considered setParentBundle

This section was denied because it was deemed to premature. An other RFC will look at nested frameworks.

setParentBundle(Bundle) – If a framework is embedded in another framework, then it must give the child framework the bundle object of its representation in the parent. That is, if you embed a framework in an OSGi framework, the parent is the bundle object of the code that manages the embedding. This bundle must be registered in the service registry as a Bundle service with the property: org.osgi.framework.parent=true. Singleton services like thread IO and URL handlers should use this service to synchronize their behavior with the ancestor frameworks. This method can be repeatedly called when the framework is not started.

# 7 Security Considerations

Obviously, a shell language provides ample opportunities for malice. In principle, anything in the system is accessible, just like from Java. The protection against malicious behavior is based up the Java 2 security model. This allows the shell and all commands to be ignorant of any security issues, unless they want to perform operations that they have access to but a potential user has not. Such code must be executed in a doPriviliged block.

The IO processors have the responsibility for protecting against malicious users.

### should copy some of the text of DMT Admin because it follows the same procedures

# 8 Document Support

## 8.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

## 8.2 Author's Address

|  |  |
|---|---|
| Name | Peter Kriens |
| Company | aQute |
| Address | 9c, Avenue St. Drezery |
| Voice | +33 633982260 |
| e-mail | Peter.Kriens@aQute.biz |

## 8.3 Acronyms and Abbreviations

## 8.4 End of Document