Command Line Interface

Draft

89 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)
New RFC	2009 April 21	Created a new RFC 147 to hold the command line interface design. RFC 132 will just be Framework Launching.
Added issues	2010 May 16	Added issues found in GOGO and POSH – Peter Kriens
Ottawa	2010 Aug 30	Processed issues reported by Derek Baum and during the Ottawa meeting

Revision	Date	Comments
Redwood	2010 Sep 16	• Package Admin and Start Level need to use adapt. Check conversion of string -> class • Converter specified to use calling do to load a class • Remove shutdown and other OSGi commands. Standard OSGI commands. Pushed to another RFC. • Made more clear that List and Map are literals and not java types • All inherited methods are considered now • Method names are first compared case sensitive and exact. if this does not provide at least 1 method, case insensitve compare is used. • Annotations are not in 1.4 so argument/parameter matching works differently. That is, flags and options also do not work pre Java 5 • Defined localization with Description annotations and a fallback with or without these annotations. • service.ranking used to determine priorty for scope when multiple scopes with the same name are registered. • Use String+ for osgi.command.function • osgi.command.description -> overrides @Description, @Description overrides resources (or uses them) • Help can work pre Java 5 using the resources • Example with map literal used [] instead of <> • Added converter from Dictionary <-> Map • my:echo = { .. }? Works with the scope? Felix has a bug #2355 • $0 is deleted • Remove child threads ... No InhertableThreadLocal • In Terminal, moved attributes to interface level • Removed ordering requirement in arguments. Sessions, named, and unnamed parameters can occur in any order. • Renamed ifPresent and ifAbsent to flag and absent in @Parameter

Revision	Date	Comments
Austin and later	14-06-11	• Parameter replaced with Option and Flag • Added Command, Description, and Scope annotations • Removed all the built-in commands • Removed all built-in variables • Moved Descriptions in separate annotations • General clean up

1 Introduction

This RFC is an solution for RFP 99 Command Provider. This RFC outlines the interfaces necessary to implement command line shells in OSG frameworks.

2 Problem Description

This RFC addresses the problem of standardized external access. The purpose of this RFC is to enable a set of basic commands that are supported by all implementations, but it must enable that bundles can provide additional commands.

2.1 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 Command Names

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

3.3 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.

3.4 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

3.5 Source Providers

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

4 Technical Solution

The designs in this section are to be part of the Compendium Specifications.

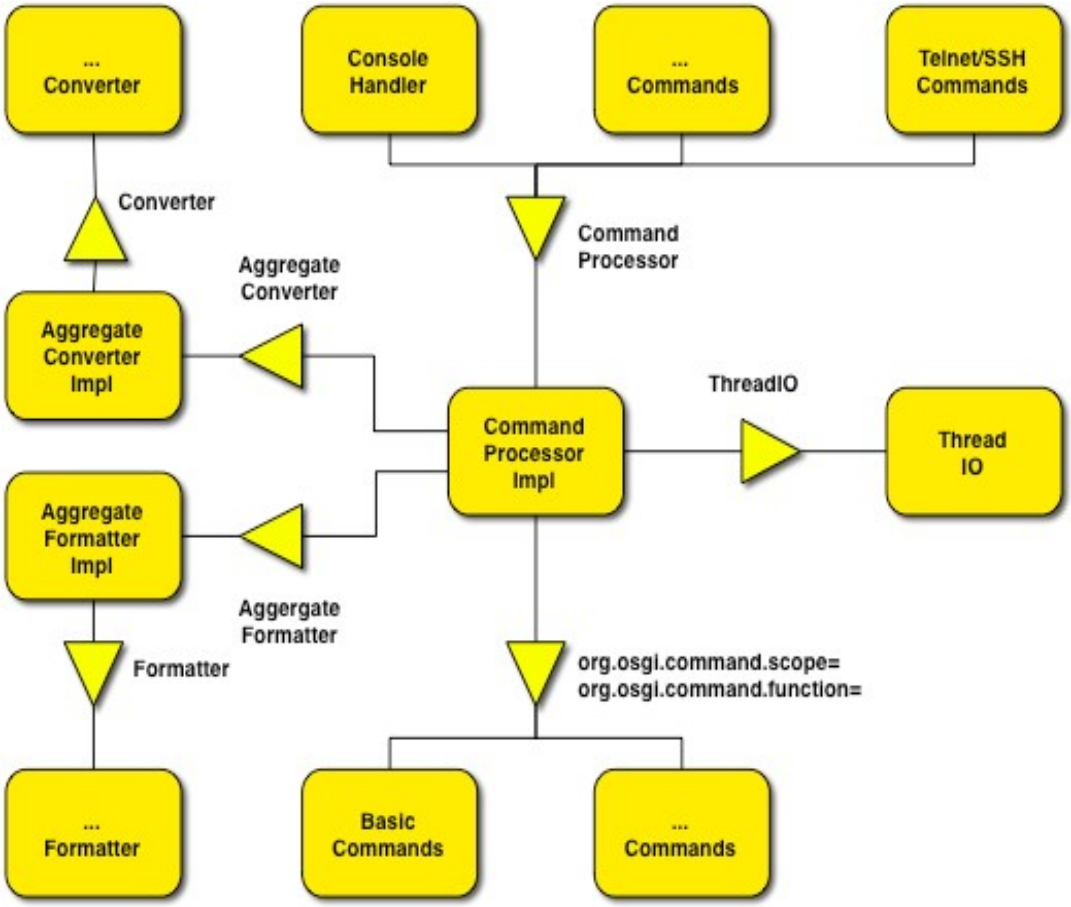
The drivers of this design have been:

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

The basic idea of the design is that there are a number of components 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. A Terminal objects is available to interact with a terminal.



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 provide streams or a Terminal service and get a Command Session in return. The session is then used 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 object → object.
- Formatter Service – Provides a facility to format objects as strings with three levels of detail.
- 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.

4.1 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 (e.g. 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, represented by a Terminal object. This allows commands to directly talk to the user, regardless if they are piped or not.
- 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.
- Meta.Scope – Provides access to meta-information about to the sub-commands of a scope. The Command Session can provide a list of Meta Scope objects.
- Terminal – An interface to a terminal allowing clients to control the cursor and attributes of the screen.It represents the input, error and output stream.
- ArgumentList – A list of arguments that is expanded when processing a command line and such an object is an argument.

4.2 Starting a Session

A user of the Command Processor must create a Command Session to use the script. 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.err, null);
Object result = session.execute("bundles|grep aQute");
String s  = new String(bout.toByteArray());
```

In this case the session is created on a set of streams. It is also possible to create a session with a Terminal. A Terminal has representations of the standard streams (well, writers and a readLine/getIn methods) but provides additional capabilities to control the screen is color, attributes and position of the cursor. If there is no terminal provided, the Command Processor service must provide a default terminal. The current terminal can be obtained with getTerminal().

4.2.1 Syntax

The tsh language requires a very simple parser to keep its footprint small. Basically a command line is broken into tokens, where the rules to parse a token are quit simple. These tokens are then each evaluated. The resulting list is then executed. Depending on patterns in this list it is either a command execution, a method call, a variable assignment, or a variable removal. Command execution and method calls use the Java type information to coerce the parameters to the target types.

Some examples:

```
$ echo Hello World
Hello World
```

The first token is the command name because it is a simple string, and is a *command*.

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 example, 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.

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 as well as reuse centralized formatting.

The printing of the result can be prevented in the following ways:

- a void command
- a null result
- if the result is assigned to a variable
- If the command has a Scope annotation and that scope indicates noprint()=true
- If the || operator is used after the command
- if the command ends with a 'semicolon (';')

Formatter services are used to print out the objects in a proper format.

4.2.2 Program Syntax and Semantics

The syntax of the shell should be simple to implement because the shell implementation 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. Also, shells like bash are very much text based while the use of actual objects is highly desirable.

An OSGi shell syntax can rely 100% on the fact that there is a Java VM. This makes it easy to control the framework and implement a shell with Java. However, a shell implies that the users directly type 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.

The syntax and dependencies are defined in an ANTLR file for now. The specification will write out those rules in more detail.By keeping this in the ANTLR it is easier to verify changes, however, there is no reason to implement an actual tsh with antlr, the format is only used for documenting.

```
grammar tsh;
@header {
package output;
}
@lexer::header {
package output;
}

//
// The pipe's have higher precedence so a program is a collection of
// pipes separated by newline or semicolons. It is allowed to
// have no pipe after the last semicolon/nl
//
tsh         : program ;
program     : pipe  (separator (pipe| ))*  ;

//
// A pipe consists of statements separated by a vertical bar. Each statement
// must be executed in a separate thread. The System.out of an earlier
// statement must be connected to the System.in of a later statement. The first
// statement has the current Terminal as input and the last statement must have
// the Terminal as output. The value of a pipe is the result of the last
// statement. If a statement that is not the last statement returns
// an object than this object must be formatted and send to System.out.
//
pipe        : statement ( piper statement )*
            ;
piper       : '|'                                   // printing pipe
            | '||'                                  // suppress printing pipe
            ;
statement   : TOKEN '=' pipe                        // Assignment local var
            | TOKEN '='                             // Remove local var
            | TOKEN '=>' pipe                       // Assignment global var
            | TOKEN '=>'                            // Remove global var
            | TOKEN value*                          // Cmd (TOKEN=scope:function)
            | value+                                // Method invocation
            ;
```

```
//
// A core aspect of tsh is the concept of a value. A value is thought of
// as a string that can be evaluated to provide an object depending on
// its type. The type is defined by its first character. The following
// types are defined:
//
value        : TOKEN                                  // Simple token
             | SSTRING                                // Single Quoted
             | DSTRING                                // Double Quoted
             | '{' program '}'                        // Closure
             | '(' program ')'                        // Evaluate
             | '[' value* ']'                         // ArrayList
             | '<' ( value '=' value ) * '>'          // LinkedHashMap
             | '$' TOKEN                              // basic macro
             | '${' TOKEN ( (':?'|':='|':+'|':-') value )? '}' // braced macro
             | '$(' program ')'                       // program macro
             | '$' ['0'..'9']                         // argument number
             | '$*'                                   // ArrayList of arguments
             | '$@'                                   // ArgumentList($*)
             ;

//
// Specifies the separator when there are multiple pipes.
//
separator   :    ';' | '\n' | COMMENT;

//
// A single or double quoted string. Double quoted strings are
// expanded with macros, single quoted not. A string can contain
// virtually any character except its start character including
// newlines. Escaping for the quotes and other escapes are supported.
//

SSTRING : '\'' (ESC_SEQ | ~('\\'|'\''))* '\''
        ;
DSTRING : '"'  (ESC_SEQ | ~('\\'|'"'))*  '"'
        ;


//
// The TOKEN is the basic element of a command line. A TOKEN is
// very liberally specified to be a sequence of one or more characters that
// is not SPECIAL to this parser.
//
TOKEN
    :    ~SPECIAL+ ;

fragment
SPECIAL : '\u0000'..'\u001F'
        | '#'
        | '|'
        | ';'
        | '='
        | '{'
```

```
        | '}'
        | '('
        | ')'
        | '$'
        | '<'
        | '>'
        | '['
        | ']'
        | ' '
        | '"'
        | '\'';

//
// Comments start with # are ignored.
//
COMMENT
    :   '#' ~('\n'|'\r')* '\r'? '\n'
    ;

//
// Whitespace is defined as spaces and tabs.
//
WS  :   ( ' ' | '\t' | '\r' )* {$channel=HIDDEN; };

//
// Escaping
// \b = backspace 08h
// \t = tab       09h
// \n = newline   0Ah
// \f = formfeed  0Ch
// \r = return    0Dh
// \\ = backslash 5Ch
// \' = squote    27h
// \" = dquote    22h
// \\n= skipped
//
fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\'|'\n')
    |   UNICODE_ESC
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;
```

The previous syntax is rather extensive to be able to describe the semantics. However, TSH can rather easily be parsed in an unstructured way. The input can be very easily tokenized, where the closure, execution, list, and map must ensure that they match their respective end tokens correctly and ensure that their start character is ignored inside strings. For example, { { { “{{{“ } } } would be a single closure. A statement is a collection of tokens without '|'

or ';' (or newline) symbols. A pipe is a collection of statements separated by the '|' symbol and a program is a set of pipes.

4.2.3 Pipes

A command can consist of a set of statements separated by the pipe symbol. This will execute the statements in different threads. The System.out of a prior pipe is the System.in of the next pipe. The first System.in is the Terminal input stream and the last System.out is the Terminal output stream.

A big question is what to do with the return value for a statement whose System.out is not the terminal. There is nobody listening for the return value and this value would get lost. In certain cases this is a problem, in other cases this is what is desired. For this reason, there are two pipe symbols:

- print ('|') - The single vertical bar symbol is the printing pipe symbol. The result of the command on the left must be formatted with the formatter using the INSPECT type to the System.out.

- suppress ('||') - A double vertical bar indicates that the result of the statement to the left of the double bars is ignored.

4.2.4 Built-In Commands

There no built-in commands.

4.2.5 Variables

There are the following variables (in order of priority):

1. Session – Variables that are set with the assignment operator ('=') always set the session variables. Session variables are local to the session and not shared with anybody else.

2. Global – The command processor maintains global variables that are shared between all sessions and should be saved persistently if possible. These variables are consulted if the session has no value for a given name. Global variables can be set with the global assignment operator ('=>'). Implementations may throw a security exception.

3. Framework – If a variable cannot be found in the session or in the global variables then the framework is asked for the variable with BundleContext.getProperty on the bundle context of the bundle that got created the session object. This will include System Properties.

Session variables provide locally added or modified commands, variables set in the command line, or variables managed by commands. Session variables are transient, they are not persistently stored, global variables that are set with the global command must be maintained persistently if they are either String or one of the primitive wrappers or a Collection of those.

There are no built-in variables.

4.2.6 Redirection

Redirection is not provided by tsh because it can be easily implemented with a command and piping. For example:

```
cmd 123 | tac somefile.txt
```

A tac command (not part of this spec) can be used to receive the output of the cmd 123 and store it in the given URI. In the same vein, a cat command could be used to provide input to a command.

There are however no built-in commands standardized.

4.2.7 Annotations

For Java 5 and later VMs, a set of annotations are available that provide help information and that provide information to for the commands and *named* parameters.

The @Scope annotation can be applied to the type and provides information about the scope name and if the object should automatically add all its public methods as commands.

The @Command annotation on a method provides a description for the command. Its noprint() field makes it possible to limit the printing of the output.

There are two types of named parameters:

- @Option – An option requires a value in the command line directly after the option alias. This value is then converted to the named parameter. The annotation can specify an optional value when the command line does not contain the option with the optional absent() field. If no absent value is specified a proper null value must be used for the parameter type. An option can be repeated multiple times if the isRepeat() field is set to true (default is false). In that case the parameters are collection in a collection before conversion is done to the parameter.

- @Flag – A flag sets a value. The annotation can specify different values for when the flag is used in a command line (value()) and when it is absent (absent()). If no absent value is specified a proper null value must be used for the parameter type. If value() or absent() is not set then the true is used when the flag is used and false when the flag is not used before conversion.

Bot named parameter annotations provide:

- alias – A list of aliases.

Examples:

```
@Command
void foo(
 @Flag( alias = "-v") boolean verbose,
 @Option( alias = {"-f", "--file"}, absent="default.txt" ) File file,
 String abc, int def
) { … }

$ foo -v 1 2   == foo(true, new File("default.txt"), "1", 2)
$ foo -f /tmp/test.txt a 8      == foo(0, new File("/tmp/test.txt"), "a", 8)
```

Some options can be repeated multiple times, this can be specified with the repeatable attribute. The parameter type must then accept conversion from a list:

```
void foo(
  @Option( alias ="-x", repeat=true ) String[] xs )
$ foo -x abc -x def -x ghi   == foo( new String[] {"abc", "def", "ghi"} )
```

It is also possible to provide default values for parameters that are not named. The Flag annotation must not specify an alias in that case and can appear anywhere in the parameters.

```
void foo(
  @Flag( absent="default") String xs )
$ foo == foo( "default" )
```

Annotations do not allow defaults of null. For this reason, the Parameter annotation has the following constants defined:

- NOT_SET – Defines that there is no value set
- NULL – Defines the value for null

These annotations only work on 1.5 and later Vms. For 1.4 Vms, each argument must be specified without options and flags.

The description and summary fields in the annotations can be localized per scope. The base name of the localization Properties resource is:

```
OSGI-INF/l10n/command.<fqn>
```

Where fqn is the fully qualified name of the object that carries the annotation. If the value of the description must use a value that starts with a % (no preceding spaces) then the percent sign is removed and the remainder of the value looked up as a key in the resources. For example:

```
@Description(value="%foo.description", summary="%foo.summary")
```

This will lookup the key foo.description.

If no description or summary field is found then the tsh must use the following default ids for the resources:

```
<method-name>.<index>.description     for a parameter description
<method-name>.description             for a method description
<method-name>.summary
description                           for the class/scope description
summary
```

If there are overloaded methods then explicit resource ids must be used in the annotation. For example:

```
@Description("%foo-0")
public void foo() {…}
@Description("%foo-1")
public void foo(String s) {…}
```

This means that without annotations it is not possible to describe overloaded methods separately.

4.2.8 Invocation

In the end, a statement consists of a collection of parsed raw *values*, this collection is called the *arguments*. The arguments consists of all parts that were presented as a *pipe* in the grammar. During invocation, at first the arguments are the raw input. An argument can have a different type depending on its lexical value:

1. TOKEN
2. Closure
3. Execution
4. STRING
5. Argument List
6. List (ArrayList)

7. Map (LinkedHashMap)
8. Macro Reference

Before a command is executed or a method invoked, the shell must look at the types of the arguments before evaluation and detect the pattern that is used for a command.There are a number of distinct cases in priority order:

1. Assignment Local - TOKEN '=' pipe
2. Delete Local - TOKEN '='
3. Assignment Global – TOKEN '=>' pipe
4. Delete Global – TOKEN '=>'
5. Command - TOKEN value*
6. Method invocation - value value value*

These patten must be detected before the raw arguments are evaluated because the evaluation will loose an important distinction: it is possible that a value is evaluated to a general string making it impossible to distinguish between them. For example, the statement `"abc" length` will map to the general method invocation while `abc length` maps to a command execution. After evaluation, they're equal.

The next step is therefore to evaluate all arguments (including the first). Evaluating has the following meaning for the different types:

1. TOKEN – Evaluates to String with its contents
2. Closure – Evaluates to an object implementing Function
3. Execution – The contents inside the parentheses are executed as a program. The resulting value is the value of the last pipe.
4. STRING – The value is the value of the string with any escaping applied. If it is a double quoted string, macros must be replaced as well.
5. Argument List – Take each member of the list as an evaluated value and add it as a new argument. This is used to implement the special behavior of $*. It allows a closure to expand its arguments for a new command. Though $* does not implement ArgumentList, the $@ macro is ArgumentList($*).
6. List – The list consists of a set of values. Each value must be evaluated and added to an ArrayList. This is a concrete class because it is the implementation used for the literal list specified with a '[' and ']'. This list is converted, if necessary, to the target type.
7. Map – The lists consists of a set of entries, where an entry is a value=value. Each entry must be evaluated and added to a LinkedHashMap.This is a concrete class because it is the implementation used for the literal map specified with a '<' and '>'. This list is converted, if necessary, to the target type.
8. Macro Reference – Macro references must be replaced with their value. See macros.

Arguments can refer to changing values, it is therefore important that the evaluation for each pipe takes place just before it is invoked. After the evaluation, the shell has a collection of evaluated arguments. For example:

```
$ foreach (bundles) {echo($it location)}
```

This command first creates the following collection of raw arguments:

1. TOKEN 'foreach'
2. Execution 'bundles'
3. Closure with value 'echo($it location)'

After evaluation, the arguments looks like:

1. String: 'foreach'
2. Bundle[]
3. Function

This collection is now executed based on the earlier detected pattern.

1. Assignment – first argument is the name of the variable. The evaluated objects after the equal sign are a pipe. This pipe must be evaluated to get the value, this value is assigned to the given variable. For example: bs = (bundles) length.
2. Removal – Remove the variable with the name of the first argument, for example: bs =
3. Assignment global -
4. Removal Global -
5. Command – The first argument in the collection is the name of the command. This name can contain a scope or not. See Finding the Command
6. Method Invocation – The first argument is the object, second is the method name. For example: $abc length.

In the case of the Command and the Invocation case it is necessary to map the remaining arguments to the parameters of the method invocation. In the case of the command call, the arguments after the command name are all method parameters because the command name implies an object and method, in the case of an invocation the values after the object and the method name are values. When a Java method is called, it is necessary to match the arguments to the method that has the best parameters to match those arguments. This requires that parameters are converted to their correct type.

4.2.9 Evaluation

Assume we have a target object t. If the execution is about a command, t is found from the command name. If the case is the invocation then t is the evaluated first value. Next the method m must be selected. In the command case, m is implied by the command name as described in provider discovery. In the invocation case m is the second evaluated value. So, $abc $def will invoke the method in $def on the object in $abc.

This means that the matching algorithm consists of finding the best method *m* on object *t* with matching name *n*, that is eligible for the given arguments.

```
Object eval( Object t, String n, List<Object> arguments )
```

1. Create a list with matching methods as given by t.getClass().getMethods() and filtered by the osgi.command.function list. This list is the set of methods whose names match the command name exactly.

2. If this list is empty, create a list with eligible methods. A method is eligible if its name matches the method name. The method name matches if:

    1. Case insensitive match with a "_" prefix. E.g. the argument is new, this will then match "_new".

    2. Case insensitive match

    3. Apply the bean design pattern to the method name and compare the name of the property in a case insensitive way. 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. The bean name is compared case insensitive.

3. Sort this list on:

    1. cardinality, higher cardinality, must be earlier in the list, if equal:

    2. parameter annotations, with parameter annotations must be earlier in the list, if equal:

    3. varargs, without varargs must be earlier in the list, if equal:

    4. canonical names of the parameter type classes, a lower name must be earlier in the list

4.2.10 Method Selection

For the method selection it is necessary to match the arguments against the parameters of the method. For this purpose, a parameter has different qualities:

- isSession – The target type is CommandSession
- isMandatory – Parameter annotation missing || absent not set
- isOption – isNamed() && flag not set
- isFlag – isNamed() && flag set
- isVarargs – last parameter of a varargs method
- isRepeat – Parameter annotation present && repeat set to true
- isNamed – Parameter annotation present && at least 1 name in alias

For each eligible method it is now necessary to see if the arguments could be mapped to this method. So for each method in sorted order:

1. Create an array for the parameters ps

2. For each argument

1. if the argument is a string and starts with “-” then
    1. remove argument from arguments
    2. if argument is “--”, break. no more parsing of named parameters in the arguments
    3. find the parameter info for the named parameter, then treat it as a serie of single character named parameters. Each of those MUST be found.
    4. Repeat the following for each parameter
        1. if a flag value is set, assign this to ps[parameter index]
        2. else remove the next argument and assign this to ps[parameter index]

The arguments have not been cleaned up of any named parameters and the results are already set in ps. For example:

```
  void foo(
   CommandSession session,
   @Flag( alias = {“-v”, “--verbose”}, value=”2”, absent=”0” ) int verbose,
   @Flag( alias = {“-x”, “--extra”}, value=”true”, absent=”false” )
                                                  boolean verbose,
   @Option( alias = {“-f”, “--file”}, absent=”default.txt” ) File file,
   String input
  );
$ foo -v abc -f x.tt
```

Initially, the arguments are: [“-v”, “-f”, “x.tt”, “abc”]. After the prior cleanup step, the result is now:

```
ps[] = new Object[5];
ps[0] = null
ps[1] = “2”
ps[2] = null
ps[3] = “x.tt”
ps[4] = null
arguments = { “abc” }
```

After this cleanup, defaults must be applied.

1. ps[last] = []
2. For each parameter index
    1. if (parameter type == CommandSession), insert it, continue
    2. if ps[index] is not set
        1. if named parameter, then use absent value. If no absent attr. set, then this is an error (mandatory)
        2. else
            1. if arguments is not empty, use next argument for ps[index]
            2. else if there is a parameter annotation with an absent value, use that one.

If arguments are left over, then this list of arguments cannot be matched to the method so this method is ignored.

In the next step, each of the values in ps[] must be converted to the appropriate parameter type using the Converter. If this conversion fails, the method is not applicable. The first method that succeeds with this process is invoked.

If no suitable method can be found and a “main(Object[])” method is available then this method must be called. The whole command line must be given, where the first parameter must be the name of the function including its scope (even if it was not specified on the command line).

If none is found, a NoSuchMethodException must be thrown.

4.2.11 Finding the Command

Finding a command must use the SCOPE variable. The SCOPE variable is a List<String> with scope names.

1. If the command name contains a scope,
    1. Look up the command name from a service as described in Command Discovery.
    2. Lookup the command as a variable in the session variables
    3. Look at the built-in commands
2. Otherwise, for each scope name in the variable SCOPE, add the scope name to the command
    1. look up the command name from a service as described in Command Discovery.
    2. Lookup the command as a variable in the session variables
    3. Look at the built-in commands
3. Lookup the command as a variable in the session variables
4. Look at the built-in commands, ignoring the prefix

4.2.12 Argument List

There is a special interface that changes the behavior of the command line processing: Argument List. A Argument List is a a List<Object>. The special behavior is that when a Argument List is used as a raw value, its content must be treated as a list of evaluated arguments. These evaluated arguments must be expanded in the list that is being processed. A normal List can be turned into an argument with the expand method. For example:

```
$ g = { grep -i $@ }  //
$ g pattern file // → grep -i pattern file
```

Any list that implements this interface will expand its contents into the array of arguments.

The $* is a normal list and will this not automatically be expanded. It is also a mutable list which allows for alternative manipulations. Its counterpart, $@, is the current $* but is expanded.

```
$ g = { $* add 0 "-i"; grep $@ }  //
$ g pattern file // → grep -i pattern file
```

4.2.13 Objects, no Strings

Arguments are not strings, they are proper objects after evaluating. Variables can refer to objects, arrays are objects, and also the result of a direct command (using parentheses ()) can result in a proper object. Matching these objects to a method is non-trivial.

4.2.14 Strings

TSH supports two types of Strings: single quoted and double quoted. Double quoted strings can contain macros. A TSH implementation must replace the macros with their value when the statement's parameters are evaluated. This must happen for every evaluation so that the macro always reflects the latest value.

There are two types of macros:

- ${ NAME (':' ('?'|'+'|'-'|'=' } - Braced macros can contain a TOKEN as variable name and will be replaced with the actual variable value.
  - ${ name : default } - The special construct allows a default value to be specified if no variable value is found.
  - ${*parameter*:?*word*} – If *parameter* is null or unset, the expansion of *word* (or a message to that effect if *word* is not present) is written to the standard error and the shell, if it is not interactive, exits. Otherwise, the value of *parameter* is substituted.
  - ${*parameter*:+*word*} – If *parameter* is null or unset, nothing is substituted, otherwise the expansion of *word* is substituted.
  - ${*parameter*:-*word*} - If *parameter* is unset or null, the expansion of *word* is substituted. Otherwise, the value of *parameter* is substituted.
  - ${*parameter*:=*word*}– If *parameter* is unset or null, the expansion of *word* is assigned to *parameter*. The value of *parameter* is then substituted. Positional parameters and special parameters may not be assigned to in this way.
- $(…) - Parentheses indicate that the macro is actually a parenthesized value and must be evaluated.

To use a string like ${ or $(in the double quoted string, escape the $ like “This is a macro: \$(...).”

4.2.15 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 a String+ property, so many names can be listed. This function name should match to a public method in the service object. If no function list is provided then the shell must get the functions by analyzing the class of the service object and using all public methods.
- osgi.command.description – A description of the scope

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);

    System.out.println(sb);
  }
}
```

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 scripts like:

```
  cfg = cm 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”
```

If multiple services register with the same scope name then they must be sorted in service.ranking order.

The Command Session's owner must be used for getting the service objects. That is, the bundle that uses the service must be the same as the bundle that got the command session.

4.2.16 Other Commands

Any other commands can be added to the shell by storing them in the session. For example:

```
  abc:def = { ... }
```

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.

4.2.17 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 the 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
```

4.2.18 Closures

The proposed specification of tsh also has *closures*. Closures represent a program that can be evaluated later. They start with a '{' and end with a recursively matching '}'. Closures can be injected as a Function in the called method.

Closures implement the Function interface. The follow code will add a command written in tsh:

```
$ my:echo = { echo xx ]$@ xx }
Closure ...
$ my:echo Hello World
xx Hello World xx
```

A closure provides a facility for delayed execution. A closure starts with a '{', it can contain a complete program, and ends with a '}'. If evaluated it will be turned into an object implementing the Function interface. A Function can be executed with parameters. A Function can be invoked many times if needed.

A closure has access to its parameters via the following macros that are only valid inside the closure:

```
$0              error
$1..$n          parameters in the given sequence, 1 is first parameter.
$it             equals to $1
$*              Mutable List<Object> that contains parameters $1 .. $n.
```

4.2.19 Help

The Command Processor has extensive information about the possible commands. This information is made available through the getMetaScopes() method on the Command Session. This method returns a snapshot of the collection of MetaScope objects. These objects are derived from the meta information of the scopes. It can come from:

1. The osgi.command.description property on a command service. A property has the highest priority for the scope description. If not set, the Description annotation or the resource information must be used.

2. The name of the methods, the return type, and the parameters method types

3. The Descriptor and Parameter annotations.

If both a property and an annotation is used, then the annotation must wins. If multiple scopes use the same name, then the descriptive texts must be concatenated with two newlines separating them.

If no annotations are used, the descriptive help information can also be found in the resources as described with the annotations.

4.2.20 Terminal

The Terminal class represents the output terminal or console. The terminal is either created by the Command Processor or, when the IO Handler passes streams end their encoding, or is created by the IO Handler. Whatever route is taken, the primary interface for IO in the shell is the Terminal.

The IO Handler can interface to many different types of terminals. From simple ASCII terminals to high level graphic windows. The capabilities can therefore differ significantly, this is reflected in the getCapabilties() method. This method returns a bitmap of capabilities.

Any terminal must support the ASCII code set and report and allow simple cursor movements. Even the most primitive ANSI terminal supports this and these movements can be emulated on even the simplest terminal.

A Terminal is associated with two Print Writer objects: System.out, System.err. Input is obtained from two methods: int getIn() and String readLine().

Normally, commands will use the System.{out,err,in} objects to do their output. It is only in special cases that the extended capabilities of the Terminal are needed. Using the Terminal will bypass any piping that was set up. The Command Processor must ensure that the System.* streams, which use the default encoding, are properly converted to the Terminal writers.

The Terminal interface provides a number of methods that can be used to control where and how the output stream ends up on the terminal. There are a number of methods that can query the terminal for its characteristics and there are a number of methods that can change the state of the terminal. Not all methods work on all terminals, the getCapabilities() method can inform the caller about any capabilities.

The terminal can report its width and height. The screen of the terminal should be see as a table with rows and columns from [0,width) x [0,height). The left-top of the screen is position 0,0 and the right bottom is width-1, height-1. Any text displayed on the screen will not wrap but will be ignored when displayed beyond the bounds. Terminals may change the last character to indicate this cut-of.

The cursor (the place where any new text on the output Print Writer will be displayed) can be positioned with tge setPosition() method, which takes an x and y coordinate. The current output Print Writer must be flushed before the position is altered. The current position can be queried with getPosition().

Terminals can support a number of attributes that alter the way the screen is displaying the text. The setAttributes(Attribute[]) method sets these attributes and return the current attributes. This is in association with the output Print Writer, not the screen position. Any subsequent writes will be displayed with the new attributes. The method returns an array of attributes which represent the previous set of attributes. Passing this array to setAttributes() will restore the previous state.

Terminals can support bi-directional text. Unicode characters have an associated direction. Terminals are not required to support bi-directional text. However, if they support bi-directional text then they must ensure that horizontal cursor movements and the movement caused by forward() and backward() are properly synchronized. That is, when the CURSOR_FORWARD is reported then forward() called, the forward method must adjust the movement depending on the direction of the underlying text. The same for CURSOR_BACKWARD and backward(). It is assumed that the underlying logic to calculate the direction is handled in a similar was as the java.text.Bidi class.

In all cases, the left-top of the screen remains 0,0 and the default progression is always left-to-right.

4.3 Thread IO Service

The Thread IO service is a framework service that guards the singletons of System.out, System.in, and System.err. It maintains a stack based model of streams per thread. Any party that wants to receive output or provide input on the current thread can push its streams. The ThreadIO service consists of two methods:

- ThreadIO.Entry push(InputStream,PrintStream,PrintStream) – Push the given 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 appropriate 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. The returned Entry must be used to pop the elements of the thread local stack. The close method must be called before the method that did the push returns.

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.Entry entry = threadio.push(null,pout,pout);
  try {
      System.out.println("Starting ...");
      doWhatever();
      System.out.println("... Done");
  } catch(Throwable t ) {
      t.printStackTrace();
  }
  finally {
    entry.close();
  }
  pout.flush();
  return new String( out.toByteArray() );
}
```

4.3.1 Child threads

Child threads created by commands must not inherit the stream from the parent thread. They must default the to the original streams.

4.3.2 Multiplexing

Multiplexing will not be in this release. Once we've decided what to do with general multiplexing then we should add this.

4.4 Converter Service

The Converter service is built around the Aggregate Converter service and the Converter services. The aggregate converter tracks any Converter services. The Aggregate Converter dispatches conversion requests to the appropriate Converter service.

The reified type is improved for the converter over blueprint. Arrays have now been defined as having a single type parameter: the component type. This approach made the conversion easier and more consistent.

The API for both services consists of (Aggregate Converter extends Converter):

- <T> T convert(Object s, ReifiedType<T> t);

The ReifiedType is defined in Blueprint.However, the type is defined in the converter package.

The Aggegrate Converter must support the following basic conversion algorithm:

1. if s == null, return null
2. Let sc = s.getClass
3. if sc isPrimitive then sc = wrapper(sc)
4. If the target type isAssignable from sc, then no conversion is necessary
5. Try all converters that indicate they can convert to the target type t in service.ranking order. The first converter that returns a non-null value succeeds the conversion
6. If the target type t == String.class, convert s to a string with toString()
7. Try converting the following basic types to each other:

->	Bool.	Byte	Char.	Short	Integ.	Long	Float	Double	BigInt.	BigDc.
Bool	v	0,1	'F','T'	0,1	0,1	0,1	0.0,1.0	0.0,1.0	0,1	0,1
Byte	v == 0 ? 0 : 1	v	v	v	v	v	v	v	v	v
Char.	v == 0 ? 0 : 1	v()	v	v	v	v	v	v	v	v
Short	v == 0 ? 0 : 1	v()	v()	v	v	v	v	v	v	v
Integ.	v == 0 ? 0 : 1	v()	v()	v()	v	v	v	v	v	v
Long	v == 0 ? 0 : 1	v()	v()	v()	v()	v	v	v	v	v
Float	v == 0 ? 0 : 1	v()	v()	v()	v()	v()	v	v	v	v
Double	v == 0 ? 0 : 1	v()	v()	v()	v()	v()	v()	v	v	v
BigInt	v == 0 ? 0 : 1	v()	v()	v()	v()	v()	v()	v()	v	v
BigDec.	v == 0 ? 0 : 1	v()	v()	v()	v()	v()	v()	v()	v	v
String	v.isEmpty ? 0 : 1	TRUE, YES, ON are true (case insens.) others are false	length == 1 , then first character	Short. parseShort	Integer. parseInt	Long. parseLong	Float. parseFloat	Double. parseDouble	new BigInteger(v)	new BigDecimal (v)

    1. v == the original value
    2. v() takes the original value and checks if it fits in the target type
8. If sc is an array type, then convert to an array
    1. Use the rules of collections to create a collection

2. Convert this to the appropriate array type, which can be a primitive array

9. If sc is assignable to Collection, convert it to a Collection<M>. M can be established through the ReifiedType t.
    1. If the t is a type listed in the replacement types, replace it with the replacement type. This creates concrete classes for abstract classes and interfaces
    2. Create a new instance of this type in l
    3. if s is an array
        1. convert each element to M and add it to l
        2. return l
    4. If sc isAssignable from Collection
        1. Convert each element to M and add it to l
        2. return l
    5. No conversion possible
10. If t isAssignable from Dictionary
    1. Create a Dictionary, if necessary using the replacement table
    2. if sc not a Map or Dictionary then conversion fails
    3. Convert s as a map or dictionary to the created dictionary while converting all keys and values.
11. If sc isAssignable to Map
    1. Create a Map, if necessary using the replacement table
    2. if sc not a Map or Dictionary then conversion fails
    3. Convert s as a map or dictionary to the created dictionary while converting all keys and values.
12. If s is not a String object then conversion fails
13. if t is an Enum class, use the valueOf method to convert s to an enum
14. Try the conversion from String to Locale, Pattern, and Properties.
15. if sc is a Class object, load the class from the string through the Bundle.loadClass method of the bundle that got the conversion service.
16. If t has a constructor that takes a String argument, use it to create a new object

Any failures in any of the previous steps should result in a failure of the conversion, a null return.

A Converter service must register with the the service property osgi.converter.classes. Its value is of type String+, reflecting the classes this converter can convert. For conversion, inheritance is not taken into account. When TSH needs to convert an object to a class, it must call an Aggregate Converter to perform the conversion. The Aggregate Converter will call the registered Converter objects in the following order:

- filtered by matching class
- sorted by service.ranking, highest first
- sorted by service.id, lowest first

4.5 Formatter Service

The Formatter service is built around the Aggregate Formatter service and the Formatter services. An Aggregate Formatter service can take an object, a level and a set of locales and return a string formatted version of this object. The Aggregate Formatter tracks any Formatter services. The Aggregate Formatter dispatches format requests to the appropriate Formatter service.

The API for both services consists of (Aggregate Formatter extends Formatter):

- CharSequence format(Object target, int level, Locale... locales) throws Exception - Format an object to a Char Sequence using the int parameter as a hint.

The hint can be INSPECT, LINE, or PART.

- INSPECT - For an INSPECT, the output can be a multiline columnar output of any reasonable level.
- LINE - A LINE format must make the object look good in a table when different objects of the same type are printed below each other.
- PART - It is allowed to use multiple lines 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.

A Formatter service must register with the the service property osgi.formatter.classes. Its value is of type String+, reflecting the classes this formatter can format. For formatting, inheritance is not taken into account. When one needs to format an object, it will call the registered Formatter objects in the following order:

- filtered by matching class
- sorted by service.ranking, highest first
- sorted by service.id, lowest first

The following code shows a simple formatter for Bundle objects:

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

```
public class BundleFormatter implements Formatter {
    BundleContext context;

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

    public CharSequence format(Object o, int level, Locale .. locales) {
        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');
    }
}
```

5 Javadoc

OSGi Javadoc

6/13/11 6:06 PM

Package Summary		Page
org.osgi.service.command	Command Package Version 1.0.	34
org.osgi.service.command.annotations	Command Annotations Package Version 1.0.	66
org.osgi.service.converter	Converter Package Version 1.0.	77
org.osgi.service.formatter	Formatter Package Version 1.0.	82
org.osgi.service.threadio	Thread IO Package Version 1.0.	86

Package org.osgi.service.command

Command Package Version 1.0.

See:
Description

Interface Summary		Page
ArgumentList	An Argument List must be expanded to its constituents when used in a command line.	35
CommandProcessor	A Command Processor is a service that that can execute a script language.	36
CommandSession	A Command Session holds the executable state of a script engine as well as a `Terminal` that is connected to that session.	38
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.	40
MetaScope	A scope defines the meta information of many commands.	41
MetaScope.MetaFunction	A Meta Function describes a scoped function.	43
MetaScope.MetaFunction.MetaParameter		45
Terminal	The Terminal interface describes a minimal terminal that can easily be mapped to command line editing tools.	50

Class Summary		Page
Terminal.Attribute	An inner class to provide an enum for the attributes	65

Exception Summary		Page
ParsingException	An exception that can point at the location where the error occurred.	48

Package org.osgi.service.command Description

Command Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.command; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.command; version="[1.0,1.1)"
```

Interface ArgumentList

org.osgi.service.command
All Superinterfaces:
Collection<Object>, Iterable<Object>, List<Object>

```
public interface ArgumentList
extends List<Object>
```

An Argument List must be expanded to its constituents when used in a command line. It allows the following pattern:

```
$ g = { grep -i (args $*) }
$ g pattern file -> grep -i pattern file
```

A ParameterList extends List, meaning that it can be manipulated as a list as well.

Interface CommandProcessor

org.osgi.service.command

public interface **CommandProcessor**

A Command Processor is a service that that can execute a script language. 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.

Version:
$Id: c2beb9d24a3416fbd6ff242e9cf9f357d352b434 $
ThreadSafe

Field Summary		Page
String	**COMMAND_DESCRIPTION** A description of the command scope.	37
String	**COMMAND_FUNCTION** A String+ of function names that may be called for this command provider.	36
String	**COMMAND_SCOPE** The scope of commands provided by this service.	36
String	**COMMAND_SUMARY** A summary of the command scope.	37

Method Summary		Page
CommandSession	**createSession**(InputStream in, PrintStream out, PrintStream err, String encoding) Create a new command session associated with IO streams.	37
CommandSession	**createSession**(Terminal terminal) Create a new Command Session that is associated with a Terminal.	37

Field Detail

COMMAND_SCOPE

public static final String **COMMAND_SCOPE** = "osgi.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. Commands can be executed as <scope>:<function>.

COMMAND_FUNCTION

public static final String **COMMAND_FUNCTION** = "osgi.command.function"

A String+ of function names that may be called for this command provider. A name may end with a *, this wildcard will then be calculated from all public methods in this service. If this property is absent but the COMMAND_SCOPE is present then all methods in the service object are used as command.

COMMAND_DESCRIPTION

```
public static final String COMMAND_DESCRIPTION = "osgi.command.description"
```

A description of the command scope. This information is available through the `org.osgi.service.command.Meta.Scope.description()` method. If this property is not set, it can come from the Description annotation or resources.

COMMAND_SUMARY

```
public static final String COMMAND_SUMARY = "osgi.command.summary"
```

A summary of the command scope. This information is available through the `org.osgi.service.command.Meta.Scope.summary()` method. If this property is not set, it can come from the Description annotation or resources.

Method Detail

createSession

```
CommandSession createSession(InputStream in,
                             PrintStream out,
                             PrintStream err,
                             String encoding)
```

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 that will be local to this session.

Parameters:

`in` - The value used for System.in. If `null` is passed, the implementation must create a valid Input Stream that always returns end of file.
`out` - The stream used for System.out, must not be `null`
`err` - The stream used for System.err, must not be `null`
`encoding` - The character encoding to use, or `null` for the default

Returns:

A new session.

---

createSession

```
CommandSession createSession(Terminal terminal)
```

Create a new Command Session that is associated with a `Terminal`. A Terminal provides the common streams but adds extra capabilities for commands to control the screen. A session maintains this Terminal.

Parameters:

`terminal` - The terminal to use in this session

Returns:

A new sessions

Interface CommandSession

org.osgi.service.command

```
public interface CommandSession
```

A Command Session holds the executable state of a script engine as well as a Terminal that is connected to that session.

Version:
$Id: 6b9a04d7a8e62aacb2d44498bdf4fd57da8e4e8c $
NotThreadSafe

Method Summary		Page
void	**close**() Close this command session.	38
Object	**execute**(CharSequence commandline) Execute a program in this session.	38
Map<String,Object>	**getGlobalVariables**() Return the Map used to maintain the global variables.	39
Collection<org.osgi.service.command.Meta.Scope>	**getMetaScopes**() Return the current list of Meta Scope objects.	39
Map<String,Object>	**getSessionVariables**() Return the modifiable Map used to store the local session variables.	39
Terminal	**getTerminal**() Return the Terminal associated with this session.	39
URI	**resolve**(String path) Resolve a local name to a URI to the current working directory.	39

Method Detail

execute

```
Object execute(CharSequence commandline)
        throws Exception,
               ParsingException
```

Execute a program in this session.

Parameters:
commandline - A Command line according to the script syntax.
Returns:
the result of the execution
Throws:
Exception - if something fails
ParsingException - If the text contained a syntax error

---

close

```
void close()
```

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

getTerminal

`Terminal` **`getTerminal`**`()`

Return the Terminal associated with this session. The terminal 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:
Terminal used closest to the user, never `null`.

getSessionVariables

`Map<String,Object>` **`getSessionVariables`**`()`

Return the modifiable Map used to store the local session variables. The returned Map can be modified to add/remove new variables, these changes are visible to the scripts running in this session.

Returns:
the map with all the variables

getGlobalVariables

`Map<String,Object>` **`getGlobalVariables`**`()`

Return the Map used to maintain the global variables. The returned map can be modified to add/remove new variables, it is thread safe.

Returns:
the Map with all the global variables

getMetaScopes

`Collection<org.osgi.service.command.Meta.Scope>` **`getMetaScopes`**`()`

Return the current list of Meta Scope objects. A scope represents a set of commands under a common name. The purpose of this information is to simplify command completion and providing extensive help about commands. If an implementation supports annotations it can use the annotations to provide extra information.

Returns:
A unmodifiable collection of `org.osgi.service.command.Meta.Scope` objects.

resolve

`URI` **`resolve`**`(String path)`

Resolve a local name to a URI to the current working directory. If the name is absolute, an absolute URI is returned.

Parameters:
`path` - A relative or absolute URI
Returns:
a URI that is the original when it was absolute and resolved against the $CWD variable if relative.

Interface Function

org.osgi.service.command

```
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. The purpose of the Function is to be injected in commands. Many commands require the possibility to execute closures or other commands. For example, a `foreach` command requires a block for execution:

```
void foreach( Iterable?> collection, Function block ) {
    for ( Object o : collection ) block.execute( Arrays.asList(o));
```

Version:
$Id: f0230553d045f15d657c30c6dc7946733f94a6de $
ThreadSafe

Method Summary		Page
Object	**execute**(CommandSession session, List<?> arguments) Execute this function and return the result.	40

Method Detail

execute

```
Object execute(CommandSession session,
               List<?> arguments)
        throws Exception
```

Execute this function and return the result.

Parameters:
`session` - The session in which to execute this function
`arguments` - The list of arguments. This list will not be modified.
Returns:
the result from the execution.
Throws:
`Exception` - if anything goes wrong

Interface MetaScope

org.osgi.service.command

```
public interface MetaScope
```

A scope defines the meta information of many commands. The information about the scopes can be obtained from CommandSession.getMetaScopes(). If annotations are present, the information is augmented with special parameter and descriptive information. Commands are stored in the session variables. Commands are added from the service registry or commands are added programmatically, however, all commands are part of the session variables and changing the session variables will change the available commands in a scope. Each command is stored in the session variables under the name: @{code :}.

Version:
$Id: af55c20f0fe35b4955a9befc1d5d6e139b7ffa50 $
ThreadSafe

Nested Class Summary		Page
static interface	**MetaScope.MetaFunction** A Meta Function describes a scoped function.	43

Method Summary		Page
String	**getDescription**() Return a description of this scope, if present.	41
Collection<MetaScope.MetaFunction>	**getMetaFunctions**() Return an unmodifiable list of Meta Function objects.	41
String	**getName**() Return the name of this scope.	41

Method Detail

getName

```
String getName()
```

Return the name of this scope.

Returns:
Name of the scope, always `non-null`.

---

getDescription

```
String getDescription()
```

Return a description of this scope, if present.

Returns:
A description or `null`

---

getMetaFunctions

```
Collection<MetaScope.MetaFunction> getMetaFunctions()
```

Return an unmodifiable list of Meta Function objects. This list is a snapshot of the current state and will not follow the state.

Returns:
an unmodifiable list of Meta Function objects.

Interface MetaScope.MetaFunction

org.osgi.service.command
Enclosing class:
MetaScope

public static interface **MetaScope.MetaFunction**

A Meta Function describes a scoped function.

Nested Class Summary		Page
static interface	**MetaScope.MetaFunction.MetaParameter**	45

Method Summary		Page
String	**getDescription**() Return a description of this scope, if present.	43
Collection<MetaScope.MetaFunction.MetaParameter>	**getMetaParameters**() Return the Meta Arguments of this Meta Function.	43
String	**getName**() Return the name of this function.	43
boolean	**isVarArgs**() Answer if this is mapped to a vararg method.	44

Method Detail

getName

String **getName**()

Return the name of this function.

Returns:
Name of the function, always non-null.

getDescription

String **getDescription**()

Return a description of this scope, if present.

Returns:
A description or null

getMetaParameters

Collection<MetaScope.MetaFunction.MetaParameter> **getMetaParameters**()

Return the Meta Arguments of this Meta Function.

Returns:
The meta arguments

---

isVarArgs

`boolean` **`isVarArgs`**`()`

Answer if this is mapped to a `vararg` method. A vararg method can be used to fill out the last argument of a function.

Returns:
`true` if this is for a vararg method, otherwise `false`

Interface MetaScope.MetaFunction.MetaParameter

org.osgi.service.command
Enclosing class:
MetaScope.MetaFunction

public static interface **MetaScope.MetaFunction.MetaParameter**

Field Summary		Page
int	**PARAMETER_FLAG** The parameter type is marked as a Flag.	46
int	**PARAMETER_OPTION** The parameter type that is marked as option.	45
int	**PARAMETER_PROVIDED_BY_SESSION** The parameter can be provided by the session, for example the Terminal object or the Command Session object.	46
int	**PARAMETER_UNNAMED** A parameter type without any meta information, a so called UNNAMED parameter.	45

Method Summary		Page
String[]	**getAliases**() List the name of aliases for this parameter.	46
String	**getDescription**() Provide a description of the parameter.	46
int	**getParameterType**() Return the Parameter Type, either: 0, PARAMETER_FLAG, PARAMETER_OPTION, PARAMETER_PROVIDED_BY_SESSION.	46
ReifiedType	**getType**() Return the Java type of the parameter.	46
String	**ifPresent**() The value to use for a flag when it is used in the command line.	47
String	**isAbsent**() The value to use when none of the parameter aliases are used in the command line for an option or a flag.	47

Field Detail

PARAMETER_UNNAMED

public static final int **PARAMETER_UNNAMED** = 0

A parameter type without any meta information, a so called UNNAMED parameter. Unnamed parameters have no Flag or Option annotation. The UNNAMED parameters must be at the end of the parameter list for a method.

PARAMETER_OPTION

public static final int **PARAMETER_OPTION** = 1

The parameter type that is marked as option. An option can be omitted from the parameter list, in that case the isAbsent() must be used. Options are marked in the command line with names that start with '-'. Options must be followed by a value, for example

```
foo -f (bar file) hello
```

PARAMETER_FLAG

```
public static final int PARAMETER_FLAG = 2
```

The parameter type is marked as a Flag. A Flag does not require a value in the command line. If the flag is used in the command line then the value is given by ifPresent(), otherwise the value is given by isAbsent().

PARAMETER_PROVIDED_BY_SESSION

```
public static final int PARAMETER_PROVIDED_BY_SESSION = 3
```

The parameter can be provided by the session, for example the Terminal object or the Command Session object.

Method Detail

getParameterType

```
int getParameterType()
```

Return the Parameter Type, either: 0, PARAMETER_FLAG, PARAMETER_OPTION, PARAMETER_PROVIDED_BY_SESSION.

Returns:
the parameter type

---

getDescription

```
String getDescription()
```

Provide a description of the parameter.

Returns:
A description of the parameter or null of none available.

---

getAliases

```
String[] getAliases()
```

List the name of aliases for this parameter. An alias must start with a minus sign ('-' -) and must not be '--'. For example: {"-f", "--file"} If the list of aliases can be empty, in that case this must be an unnamed parameter.

Returns:
Array of aliases.

---

getType

```
ReifiedType getType()
```

Return the Java type of the parameter. This is a Reified Type that can be used in the conversion model.

Returns:
the refied type that describes the parameter's type.

---

isAbsent

```
String isAbsent()
```

The value to use when none of the parameter aliases are used in the command line for an option or a flag.

Returns:
The value to use when the parameter is not used in the command line, can be `null`.

---

ifPresent

```
String ifPresent()
```

The value to use for a flag when it is used in the command line. This method is undefined for an option.

Returns:
the parameter value for a flag.

Class ParsingException

org.osgi.service.command

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ org.osgi.service.command.ParsingException
```

All Implemented Interfaces:
Serializable

```
public class ParsingException
extends RuntimeException
```

An exception that can point at the location where the error occurred.

Constructor Summary	Page
ParsingException(String message, int line, int column) Constructor to create a syntax exception.	48

Method Summary		Page
int	getColumn() Get the column number where the parser failed.	49
int	getLine() Get the line number where the parser failed.	48

Constructor Detail

ParsingException

```
public ParsingException(String message,
                        int line,
                        int column)
```

Constructor to create a syntax exception.

Parameters:
`message` - A human readable message
`line` - The line number at which the parser failed to recognize the text
`column` - The column at which the parser failed to recognize the text

Method Detail

getLine

```
public int getLine()
```

Get the line number where the parser failed.

Returns:
The line # where the parser failed.

---

getColumn

```
public int getColumn()
```

Get the column number where the parser failed.

Returns:
The column # where the parser failed.

Interface Terminal

org.osgi.service.command

public interface **Terminal**

The Terminal interface describes a minimal terminal that can easily be mapped to command line editing tools. A Terminal is associated with an Input Stream and an Output Stream. The Input Stream represents the keyboard and the Output Stream the screen. A terminal does not block the input, each character is returned as it is typed, no buffering or line editing takes place, characters are also not echoed. However, the Input Stream is not restricted to bytes only, it can also return translated key strokes. Integers from 1000 are used for those. Not all keys have to be supported by an implementation. A number of functions is provided to move the cursor and erase characters/lines/screens. Any text outputed to the Output Stream is immediately added to the cursor position, which is then moved forwards. The control characters (LF,CR,TAB,BS) perform their normal actions. However lines do not wrap. Text typed that is longer than the window will not be visible, it is the responsibility of the sender to ensure this does not happen. A screen is considered to be `getHeight()` lines that each have `getWidth()` characters. For cursor positioning, the screen is assumed to be starting at 0,0 and increases its position from left to right and from top to bottom. Positioning outside the screen bounds is undefined.

Nested Class Summary		Page
static class	**Terminal.Attribute** An inner class to provide an enum for the attributes	65

Field Summary		Page
Terminal.Attribute	**BACK_BLACK** Black	58
Terminal.Attribute	**BACK_BLUE** Blue	59
Terminal.Attribute	**BACK_CYAN** Cyan	59
Terminal.Attribute	**BACK_DEFAULT** Default background color.	58
Terminal.Attribute	**BACK_GREEN** Green	58
Terminal.Attribute	**BACK_MAGENTA** Magenta	58
Terminal.Attribute	**BACK_NONE** No Color, transparent.	58
Terminal.Attribute	**BACK_RED** Red	59
Terminal.Attribute	**BACK_WHITE** White	59
Terminal.Attribute	**BACK_YELLOW** Yellow	58
Terminal.Attribute	**BOLD** Bolden the text.	56
int	**BREAK** Break key	54
int	**CONTROL_START** Start point of control characters.	53
int	**CURSOR_BACKWARD** Cursors backward key.	53
int	**CURSOR_DOWN** Cursor down key.	53

int	**CURSOR_FORWARD** Cursors forward key.	53
int	**CURSOR_UP** Cursor up key	53
int	**DELETE** Delete key	54
int	**END** End key	54
int	**F1** Function key 1	55
int	**F10** Function key 10	56
int	**F11** Function key 11	56
int	**F12** Function key 12	56
int	**F2** Function key 2	55
int	**F3** Function key 3	55
int	**F4** Function key 4	55
int	**F5** Function key 5	55
int	**F6** Function key 6	55
int	**F7** Function key 7	55
int	**F8** Function key 8	56
int	**F9** Function key 9	56
Terminal.Attribute	**FORE_BLACK** Black	57
Terminal.Attribute	**FORE_BLUE** Blue	57
Terminal.Attribute	**FORE_CYAN** Cyan	57
Terminal.Attribute	**FORE_DEFAULT** Default foreground color.	57
Terminal.Attribute	**FORE_GREEN** Green	57
Terminal.Attribute	**FORE_MAGENTA** Magenta	57
Terminal.Attribute	**FORE_NONE** No Color, transparent.	57
Terminal.Attribute	**FORE_RED** Red	58
Terminal.Attribute	**FORE_WHITE** White	58
Terminal.Attribute	**FORE_YELLOW** Yellow	57
int	**FUNCTION_START** Helper	55

int	**HOME** Home key	54
int	**INSERT** Insert key	54
int	**PAGE_DOWN** Page down key	54
int	**PAGE_UP** Page up key	54
long	**REPORTS_CURSOR_POS** Value for getCapabilities(), if set this Terminal can report the cursor position.	59
long	**REPORTS_SIZE** Value for getCapabilities(), if set this Terminal can report the current size, see getHeight() and getWidth().	59
long	**REPORTS_SIZE_CHANGES** Value for getCapabilities(), if set this Terminal sends a control code when the terminal changes size.	59
Terminal.Attribute	**REVERSED** Reverse the text.	56
int	**SIZE_CHANGE** The window size has changed.	54
long	**SUPPORTS_ATTRIBUTES** Value for getCapabilities(), if set this Terminal supports attributes.	60
long	**SUPPORTS_BIDIRECTIONAL_SCRIPTS** Value for getCapabilities(), if set this Terminal will handle bidirectional scripts.	59
long	**SUPPORTS_CURSOR_POS** Value for getCapabilities(), if set this Terminal supports setting the cursor position.	60
long	**SUPPORTS_LATIN_1_SUPPLEMENT** Value for getCapabilities(), if set this Terminal supports the LATIN_1 supplement.	60
long	**SUPPORTS_UNICODE** Value for getCapabilities(), if set this Terminal supports the full UNICODE set.	60
Terminal.Attribute	**UNDERLINE** Underline the text.	56

Method Summary

		Page
Terminal.Attribute[]	**attributes**(Terminal.Attribute... attr) Set the attributes of the text to outputed next.	63
void	**backward**() Move the cursor backward.	62
void	**clear**() Clear the complete screen and position the cursor at 0,0.	61
void	**down**() Move the cursor down one line, this must not cause a scroll if the cursors moves off the screen.	62
void	**eraseEndOfLine**() Leave the cursor where it is but clear the remainder of the line.	61
void	**forward**() Move the cursor forward.	62
long	**getCapabilities**() Answer the capabilities of this terminal.	64
PrintWriter	**getErr**() Return the associated standard error stream.	61
int	**getHeight**() Return the actual height of the screen.	62

int	**getIn**() Get a character from the input.	*60*
PrintWriter	**getOut**() Return the associated standard output stream.	*61*
int[]	**getPosition**() Return the current cursor position.	*63*
int	**getWidth**() Return the actual width of the screen.	*62*
boolean	**position**(int x) Position the cursor x position on the screen.	*63*
boolean	**position**(int x, int y) Position the cursor on the screen.	*63*
String	**readLine**() Read a complete line from the input.	*60*
void	**up**() Move the cursor up one line, this must not cause a scroll if the cursor moves off the screen.	*61*

Field Detail

CONTROL_START

```
public static final int CONTROL_START = 65536
```

Start point of control characters.

CURSOR_UP

```
public static final int CURSOR_UP = 65536
```

Cursor up key

CURSOR_DOWN

```
public static final int CURSOR_DOWN = 65537
```

Cursor down key.

CURSOR_FORWARD

```
public static final int CURSOR_FORWARD = 65538
```

Cursors forward key. Usually right.

CURSOR_BACKWARD

```
public static final int CURSOR_BACKWARD = 65539
```

Cursors backward key. Usually left.

PAGE_UP

```
public static final int PAGE_UP = 65540
```

Page up key

PAGE_DOWN

```
public static final int PAGE_DOWN = 65541
```

Page down key

HOME

```
public static final int HOME = 65542
```

Home key

END

```
public static final int END = 65543
```

End key

INSERT

```
public static final int INSERT = 65544
```

Insert key

DELETE

```
public static final int DELETE = 65545
```

Delete key

BREAK

```
public static final int BREAK = 65546
```

Break key

SIZE_CHANGE

```
public static final int SIZE_CHANGE = 65547
```

The window size has changed.

FUNCTION_START

```
public static final int FUNCTION_START = 65792
```

Helper

F1

```
public static final int F1 = 65793
```

Function key 1

F2

```
public static final int F2 = 65794
```

Function key 2

F3

```
public static final int F3 = 65795
```

Function key 3

F4

```
public static final int F4 = 65796
```

Function key 4

F5

```
public static final int F5 = 65797
```

Function key 5

F6

```
public static final int F6 = 65798
```

Function key 6

F7

```
public static final int F7 = 65799
```

Function key 7

F8

```
public static final int F8 = 65800
```

Function key 8

F9

```
public static final int F9 = 65801
```

Function key 9

F10

```
public static final int F10 = 65802
```

Function key 10

F11

```
public static final int F11 = 65803
```

Function key 11

F12

```
public static final int F12 = 65804
```

Function key 12

UNDERLINE

```
public static final Terminal.Attribute UNDERLINE
```

Underline the text.

BOLD

```
public static final Terminal.Attribute BOLD
```

Bolden the text.

REVERSED

```
public static final Terminal.Attribute REVERSED
```

Reverse the text.

FORE_DEFAULT

public static final Terminal.Attribute **FORE_DEFAULT**

Default foreground color.

FORE_NONE

public static final Terminal.Attribute **FORE_NONE**

No Color, transparent.

FORE_BLACK

public static final Terminal.Attribute **FORE_BLACK**

Black

FORE_GREEN

public static final Terminal.Attribute **FORE_GREEN**

Green

FORE_YELLOW

public static final Terminal.Attribute **FORE_YELLOW**

Yellow

FORE_MAGENTA

public static final Terminal.Attribute **FORE_MAGENTA**

Magenta

FORE_CYAN

public static final Terminal.Attribute **FORE_CYAN**

Cyan

FORE_BLUE

public static final Terminal.Attribute **FORE_BLUE**

Blue

FORE_RED

public static final Terminal.Attribute **FORE_RED**

Red

FORE_WHITE

public static final Terminal.Attribute **FORE_WHITE**

White

BACK_DEFAULT

public static final Terminal.Attribute **BACK_DEFAULT**

Default background color.

BACK_NONE

public static final Terminal.Attribute **BACK_NONE**

No Color, transparent.

BACK_BLACK

public static final Terminal.Attribute **BACK_BLACK**

Black

BACK_GREEN

public static final Terminal.Attribute **BACK_GREEN**

Green

BACK_YELLOW

public static final Terminal.Attribute **BACK_YELLOW**

Yellow

BACK_MAGENTA

public static final Terminal.Attribute **BACK_MAGENTA**

Magenta

BACK_CYAN

```
public static final Terminal.Attribute BACK_CYAN
```

Cyan

BACK_BLUE

```
public static final Terminal.Attribute BACK_BLUE
```

Blue

BACK_RED

```
public static final Terminal.Attribute BACK_RED
```

Red

BACK_WHITE

```
public static final Terminal.Attribute BACK_WHITE
```

White

REPORTS_CURSOR_POS

```
public static final long REPORTS_CURSOR_POS = 1L
```

Value for getCapabilities(), if set this Terminal can report the cursor position. See position(int, int)

REPORTS_SIZE_CHANGES

```
public static final long REPORTS_SIZE_CHANGES = 2L
```

Value for getCapabilities(), if set this Terminal sends a control code when the terminal changes size. See SIZE_CHANGE.

REPORTS_SIZE

```
public static final long REPORTS_SIZE = 4L
```

Value for getCapabilities(), if set this Terminal can report the current size, see getHeight() and getWidth().

SUPPORTS_BIDIRECTIONAL_SCRIPTS

```
public static final long SUPPORTS_BIDIRECTIONAL_SCRIPTS = 256L
```

Value for getCapabilities(), if set this Terminal will handle bidirectional scripts. If supported, text and cursor movements must be automatically reordered to match the visualization. This will allow users to just send Unicode strings where text is in increasing memory order. Any reordering is only done on the display.

SUPPORTS_ATTRIBUTES

```
public static final long SUPPORTS_ATTRIBUTES = 1024L
```

Value for getCapabilities(), if set this Terminal supports attributes. If not set, attributes(Attribute...) will always return null.

SUPPORTS_CURSOR_POS

```
public static final long SUPPORTS_CURSOR_POS = 16384L
```

Value for getCapabilities(), if set this Terminal supports setting the cursor position. If not set, position(int, int) will always return false and not set the cursor.

SUPPORTS_LATIN_1_SUPPLEMENT

```
public static final long SUPPORTS_LATIN_1_SUPPLEMENT = 65536L
```

Value for getCapabilities(), if set this Terminal supports the LATIN_1 supplement. These are the UNICODE 80-FF codes. A Terminal must minimally support US-ASCII.

SUPPORTS_UNICODE

```
public static final long SUPPORTS_UNICODE = 131072L
```

Value for getCapabilities(), if set this Terminal supports the full UNICODE set. This does not imply Bidirectional script handling. A Terminal must minimally support US-ASCII.

Method Detail

getIn

```
int getIn()
          throws Exception
```

Get a character from the input. Characters less than 0x10000 are Unicode characters, if more it is a control code defined by the constants in this class.

Returns:
the current Input Stream.
Throws:
Exception - When character cannot be read

---

readLine

```
String readLine()
         throws Exception
```

Read a complete line from the input. The result will not contain any command codes, just text. Implementers can allow line editing and history handling. The string must not contain the LF or CR at the end.

Returns:
a new line
Throws:
`Exception`

---

getOut

```
PrintWriter getOut()
```

Return the associated standard output stream.

Returns:
the associated standard output stream

---

getErr

```
PrintWriter getErr()
```

Return the associated standard error stream.

Returns:
the associated standard error stream

---

clear

```
void clear()
    throws Exception
```

Clear the complete screen and position the cursor at 0,0.

Throws:
`Exception` - when the method fails

---

eraseEndOfLine

```
void eraseEndOfLine()
            throws Exception
```

Leave the cursor where it is but clear the remainder of the line.

Throws:
`Exception` - when the method fails

---

up

```
void up()
        throws Exception
```

Move the cursor up one line, this must not cause a scroll if the cursor moves off the screen.

Throws:
`Exception` - when the method fails

---

down

```
void down()
          throws Exception
```

Move the cursor down one line, this must not cause a scroll if the cursors moves off the screen.

Throws:
`Exception` - when the method fails

---

backward

```
void backward()
       throws Exception
```

Move the cursor backward. Must not wrap to previous line.

Throws:
`Exception` - when the method fails

---

forward

```
void forward()
      throws Exception
```

Move the cursor forward. Must not wrap to next line if the cursor becomes higher than the width.

Throws:
`Exception` - when the method fails

---

getWidth

```
int getWidth()
      throws Exception
```

Return the actual width of the screen. Some screens can change their size and this method must return the actual width if possible. If the width cannot be established a -1 must be returned. If the size changes and the terminal supports reporting these events a `SIZE_CHANGE` key must be returned.

Returns:
the width of the screen or -1.
Throws:
`Exception` - when the method fails

---

getHeight

```
int getHeight()
       throws Exception
```

Return the actual height of the screen. Some screens can change their size and this method must return the actual height if possible. If the width cannot be established a -1 must be returned. If the size changes and the terminal supports reporting these events a `SIZE_CHANGE` key must be returned.

Returns:
the height of the screen or -1.
Throws:
`Exception` - when the method fails

---

getPosition

```
int[] getPosition()
           throws Exception
```

Return the current cursor position. The position is returned as an array of 2 elements. The first element is the x position and the second elements is the y position. Both are zero based.

Returns:
the current position or `null` if it is not possible to establish the cursor position.
Throws:
`Exception` - when the method fails

---

position

```
boolean position(int x,
                 int y)
          throws IllegalArgumentException,
                 Exception
```

Position the cursor on the screen. Positioning starts at 0,0 and the maximum value is given by `getWidth()`, `getHeight()`. The visible cursor is moved to this position and text insertion will continue from that position.

Parameters:
`x` - The x position, must be from 0-width
`y` - The y position, must be from 0-height
Returns:
`true` if the position could be set, otherwise `false`
Throws:
`IllegalArgumentException` - when x or y is not in range
`Exception` - when this method fails

---

position

```
boolean position(int x)
          throws IllegalArgumentException,
                 Exception
```

Position the cursor x position on the screen. Is the same as position(x,y), where y represents the current line on the screen.

Parameters:
`x` - The x position, must be from 0-width
Returns:
`true` if the position could be set, otherwise `false`
Throws:
`IllegalArgumentException` - when x or y is not in range
`Exception` - when this method fails

---

attributes

```
Terminal.Attribute[] attributes(Terminal.Attribute... attr)
                        throws Exception
```

Set the attributes of the text to outputed next. The method returns the current settings, which can be used to restore the display to the previous state. These current settings are stream based and not associated with the position of the cursor. Attributes must be completely specified, they do not inherit from the current display. If attributes are specified multiple times, the last one wins.

Parameters:
`attr` - A number of attributes describing the output
Returns:
The previous state of attributes or null if attributes are not supported.
Throws:
`Exception` - when this method fails

---

getCapabilities

`long` **`getCapabilities`**`()`

Answer the capabilities of this terminal. The following capabilities can be returned:

Returns:
the bitmap of capabilities

Class Terminal.Attribute

org.osgi.service.command

```
java.lang.Object
  └ org.osgi.service.command.Terminal.Attribute
```

Enclosing class:
Terminal

```
public static class Terminal.Attribute
extends Object
```

An inner class to provide an enum for the attributes

Method Summary		Page
String	toString()	65

Method Detail

toString

```
public String toString()
```

Overrides:
toString in class Object

Package org.osgi.service.command.annotations

Command Annotations Package Version 1.0.

See:
Description

Annotation Types Summary		Page
Command	Indicates that this	67
Description	Provide a description for this element.	68
Flag	A Flag provides the information to treat a method parameter as a flag.	69
Option	An Option provides the information to treat a method parameter an option with a value.	71
Parameter	A Parameter provides the information to treat a method parameter as a flag or option.	73
Scope	Indicates that this	75

Package org.osgi.service.command.annotations Description

Command Annotations Package Version 1.0.

The purpose of this package is to provide access to the annotations for the command shell.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.command.annotations; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.command.annotations; version="[1.0,1.1)"
```

Annotation Type Command

org.osgi.service.command.annotations

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value=ElementType.METHOD)
public @interface Command
```

Indicates that this

Version:
$Id: aae39c34b715908cd0317ec725b8680ff7641300 $

Required Element Summary		Page
String	**description** The descriptive text.	67
boolean	**noprint** Ensure the result is never printed.	67
String	**summary**	67

Element Detail

description

```
public abstract String description
```

The descriptive text.

Returns:
the descriptive text

summary

```
public abstract String summary
```

Default:
""

Returns:
the summary or null

noprint

```
public abstract boolean noprint
```

Ensure the result is never printed.

Default:
false

Annotation Type Description

org.osgi.service.command.annotations

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value={
  ElementType.TYPE,
  ElementType.METHOD,
  ElementType.PARAMETER
})
public @interface Description
```

Provide a description for this element. This is a generic annotation that can be used describe types, methods, and parameters. The information in this annotation can end up in the Command Descriptor. The usage of the Description is like:

Version:
$Id: c97161ad7adc294d6eb706e182afb0d547d18538 $

Required Element Summary		Page
String	**value** The descriptive text.	68

Element Detail

value

```
public abstract String value
```

The descriptive text.

Returns:
the descriptive text

Annotation Type Flag

org.osgi.service.command.annotations

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value=ElementType.PARAMETER)
public @interface Flag
```

A Flag provides the information to treat a method parameter as a flag. Flags always start with a minus sign ('-, -) in the command line, a flag goes without a value. If the same name aliases are used in other Flags or Options than the first one in the parameter declaration wins. Flags and Options must be the first parameters in the method, it is not possible to intersperse the flags and options with parameters that have none. All remaining parameters without a Flag or Option description are unnamed.

Version:
$Id: 3a9088cb50eadcf3499512c857b6b9bd58e4bfc9 $

Field Summary		Page
String	**NOT_SET** Magic value to indicate that the absent value is not set.	69
String	**NULL** Magic value to indicate that a value is supposed to be null.	69

Required Element Summary		Page
String	**absent** The value of the parameter if its name is absent on the command line.	70
String[]	**alias** Parameter name and aliases.	69
String	**description** The descriptive text.	70
String	**value** The value if the flag is used in the command line.	70

Field Detail

NOT_SET

```
public static final String NOT_SET = "326285af7359e197efbeac04ef9e4443a3ea1281"
```

Magic value to indicate that the absent value is not set. The shell must then use null,0, or false depending on the type.

NULL

```
public static final String NULL = "aa0b23939aa1ddf22f5d8d7312968f602d8100b3"
```

Magic value to indicate that a value is supposed to be null.

Element Detail

alias

```
public abstract String[] alias
```

Parameter name and aliases. The shell will only be able to recognize names that start with a hyphen ('-', '-'), however, other names are allowed. If the list of aliases is empty, For example, to support the options -f/--files, return {"-f", "--files"}. If the alias contains an empty array then the Parameter can be provided by the session because it is a built in value like the Command Session object or the Terminal.

Returns:
parameter names.

---

value

```
public abstract String value
```

The value if the flag is used in the command line. If the flag is not used in the command line then the absent value is used.

Default:
"326285af7359e197efbeac04ef9e4443a3ea1281"
Returns:
The value to use when one of the flag names is present in the command line.

---

absent

```
public abstract String absent
```

The value of the parameter if its name is absent on the command line.

Default:
"326285af7359e197efbeac04ef9e4443a3ea1281"
Returns:
default value of the parameter if its name is not present on the command line.

---

description

```
public abstract String description
```

The descriptive text.

Returns:
the descriptive text

Annotation Type Option

org.osgi.service.command.annotations

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value=ElementType.PARAMETER)
public @interface Option
```

An Option provides the information to treat a method parameter an option with a value. Options and flags always start with a minus sign ('-, -) in the command line. An option always is followed by a value. If the same name alias is used by other Flags or Options then the first one in the parameter declaration wins. Flags and Options must be the first parameters in the method, it is not possible to intersperse the Flags and Options with parameters that have none. All remaining parameters without a Parameter description are unnamed.

Version:
$Id: 165409ae97a66ad2ae4d70dff308322c13f79276 $

Field Summary		Page
String	**NOT_SET** Magic value to indicate that a value is not set.	71
String	**NULL** Magic value to indicate that a value is supposed to be null.	71

Required Element Summary		Page
String	**absent** The value of the parameter if its name is not present on the command line.	72
String[]	**alias** Parameter name and aliases.	71
String	**description** The descriptive text.	72
boolean	**repeat** Set if this parameter can be repeated multiple times.	72

Field Detail

NOT_SET

```
public static final String NOT_SET = "326285af7359e197efbeac04ef9e4443a3ea1281"
```

Magic value to indicate that a value is not set. The shell must then use null,0, or false depending on the type.

NULL

```
public static final String NULL = "aa0b23939aa1ddf22f5d8d7312968f602d8100b3"
```

Magic value to indicate that a value is supposed to be null.

Element Detail

alias

```
public abstract String[] alias
```

Parameter name and aliases. The shell will only be able to recognize names that start with a hyphen ('-', '-'), however, other names are allowed. If the list of aliases is empty, For example, to support the options -f/--files, return {"-f", "--files"}. If the alias contains an empty array then the Parameter can be provided by the session because it is a built in value like the Command Session object or the Terminal.

Default:
{}
Returns:
parameter names.

---

absent

```
public abstract String absent
```

The value of the parameter if its name is not present on the command line. This value is effectively the default value for the parameter. If this method returns NOT_SET then this option must be specified.

Default:
"326285af7359e197efbeac04ef9e4443a3ea1281"
Returns:
default value of the parameter if its name is not present on the command line.

---

repeat

```
public abstract boolean repeat
```

Set if this parameter can be repeated multiple times. For an option, the collected parameters will be aggregated in a Collection.

Default:
false

---

description

```
public abstract String description
```

The descriptive text.

Returns:
the descriptive text

Annotation Type Parameter

org.osgi.service.command.annotations

```
public @interface Parameter
```

A Parameter provides the information to treat a method parameter as a flag or option. Options and flags always start with a minus sign ('-, -) in the command line. An option always is followed by a value while a flag goes without a value. The distinction is made with the ifPresent() method. If this method returns NOT_SET then this Parameter describes an option, otherwise it is a flag because a value is provided when the flag is used. If the same name aliases are used in other flags or options than the first one in the parameter declaration wins. Flags and options must be the first parameters in the method, it is not possible to intersperse the flags and options with parameters that have none. All remaining parameters without a Parameter description are unnamed.

Version:
$Id: 9e89771bf12143b1d532b6c09bcab8669d9cd26b $

Field Summary		Page
String	**NOT_SET** Magic value to provide a default for ifAbsent() and ifPresent() to indicate it is not set.	73

Required Element Summary		Page
String[]	**alias** Parameter name and aliases.	73
String	**ifAbsent** The value of the parameter if its name is not present on the command line.	74
String	**ifPresent** The value if the flag or option is used in the command line.	74

Field Detail

NOT_SET

```
public static final String NOT_SET = "be3831f6ddfaa48efe1c0aba9e81c6251bf0f0ca"
```

Magic value to provide a default for ifAbsent() and ifPresent() to indicate it is not set.

Element Detail

alias

```
public abstract String[] alias
```

Parameter name and aliases. The shell will only be able to recognize names that start with a hyphen ('-', '-'), however, other names are allowed. If the list of aliases is empty, For example, to support the options -f/--files, return {"-f", "--files"}. If the alias contains an empty array then the Parameter can be provided by the session because it is a built in value like the Command Session object or the Terminal.

Returns:
parameter names.

ifPresent

```
public abstract String ifPresent
```

The value if the flag or option is used in the command line. If this returns NOT_SET then this is an option and the value after the alias must be used. This value mist always be set for a flag.

Default:
"be3831f6ddfaa48efe1c0aba9e81c6251bf0f0ca"
Returns:
The value to use when one of the flag names is present in the command line.

---

ifAbsent

```
public abstract String ifAbsent
```

The value of the parameter if its name is not present on the command line. This value is effectively the default value for the parameter. If this method returns NOT_SET then an appropriate default must be chosen that is negative. That is, 0, false, null.

Returns:
default value of the parameter if its name is not present on the command line.

Annotation Type Scope

org.osgi.service.command.annotations

```
@Retention(value=RetentionPolicy.RUNTIME)
@Target(value=ElementType.TYPE)
public @interface Scope
```

Indicates that this

Version:
$Id: 22494cd81c7c6ab983b9c9917a61013d7d5578d0 $

Required Element Summary		Page
boolean	**auto** Automatically use all public methods as commands.	75
String	**description** The descriptive text.	75
String	**scope** The scope name for this type.	75
String	**summary**	76

Element Detail

scope

```
public abstract String scope
```

The scope name for this type.

Returns:
the scope name

auto

```
public abstract boolean auto
```

Automatically use all public methods as commands.

Default:
false
Returns:
true if this scope should be analyzed for commands.

description

```
public abstract String description
```

The descriptive text.

Returns:
the descriptive text

summary

```
public abstract String summary
```

Default:
""

Returns:
the summary or null

Package org.osgi.service.converter

Converter Package Version 1.0.

See:
[Description]

Interface Summary		Page
AggregateConverter	The Aggregate Converter service aggregates the Converter services present in the service registry.	*78*
Converter	A Converter service can convert an object to another *type*.	*79*

Class Summary		Page
ReifiedType	Provides access to a concrete type and its optional generic type parameters without relying on the Java 5 (and later) Type classes.	*80*

Package org.osgi.service.converter Description

Converter Package Version 1.0.

The purpose of this package is to provide access to a Converter service.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.converter; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.converter; version="[1.0,1.1)"
```

Interface AggregateConverter

org.osgi.service.converter
All Superinterfaces:
Converter

```
public interface AggregateConverter
extends Converter
```

The Aggregate Converter service aggregates the Converter services present in the service registry. That is, a Converter service provides the converting facility for one or more types, an Aggregate Converter service provides converting for any object by using the Converter services present in the service registry, as well as providing a number of basic type conversions. When called, an Aggregate Converter must use the Converter.OSGI_CONVERTER_TYPE property of the Converter services to determine which service to use. Finding the right converter must follow the following rules: The goal of the type conversion is to convert a source object `s` with type `S` to a target type `T<P1..Pn>`. The Aggregator Converter must attempt to find the first Converter service (in service.ranking order) that successfully performs this conversion. If no capable Converter service can be found, the Aggregator Converter service must provide a number of basic conversions as described in the specification. These conversions handle arrays, collections, and many built-in types, see the specification. An Aggregate Formatter must not register with a Converter.OSGI_CONVERTER_TYPE service property. The Aggregate Converter service uses the same signature as the Converter Service, this interface is therefore a marker interface to simplify getting access to the Aggregator service.

Version:
$Id: ac1c6a698bebc74fd554c1c2cc9cd1afbd807c1a $
ThreadSafe

Fields inherited from interface org.osgi.service.converter.Converter
OSGI_CONVERTER_TYPE

Methods inherited from interface org.osgi.service.converter.Converter
convert

Interface Converter

org.osgi.service.converter
All Known Subinterfaces:
AggregateConverter

```
public interface Converter
```

A Converter service can convert an object to another *type*. For example the command shell can use this service to coerce the source type (in this case often a string) to a destination type, for example a parameter in a method. However, converters are general converters, they must be able to handle any source type though it is not required that they can actually perform the conversion. A Converter service must register with the OSGI_CONVERTER_TYPE service property. This property specifies the names of the class and interfaces this Converter service can convert to one or more target types. Multiple Converter services can provide conversion of the same class/interface. An AggregateConverter must use these services in service ranking order, the first Converter service that can convert the source object to the target type will be chosen. Due to different class space the target type can be a recognized class name but still fail because the converter belongs to another class space. Converter services must therefore be careful to check if they are in the same class space as the target type.

Version:
$Id: 205388b7fbf30e23ff0bc84bd53af30c213d6964 $

Field Summary		Page
String	**OSGI_CONVERTER_TYPE** Names of classes/interfaces that this Converter service has conversions for.	79

Method Summary		Page
T	**convert**(Object sourceObject, ReifiedType<T> targetType) Convert an object to the desired type.	79

Field Detail

OSGI_CONVERTER_TYPE

```
public static final String OSGI_CONVERTER_TYPE = "osgi.converter.type"
```

Names of classes/interfaces that this Converter service has conversions for. It is a service property that is a `String+`. That is, a string, or an array/collection of strings.

Method Detail

convert

```
T convert(Object sourceObject,
          ReifiedType<T> targetType)
```

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

Type Parameters:
`T` - The raw type of the desired result object
Parameters:
`sourceObject` - The object that must be converted
`targetType` - The type that the returned object can be assigned to
Returns:
An object that can be assigned to the desired type or `null` if it could not be converted.

Class ReifiedType

org.osgi.service.converter

```
java.lang.Object
  └ org.osgi.service.converter.ReifiedType
```

Type Parameters:
`T` - The raw type.

```
public class ReifiedType
extends Object
```

Provides access to a concrete type and its optional generic type parameters without relying on the Java 5 (and later) Type classes. Java 5 and later support generic types. These types consist of a *raw* class with type parameters, e.g. `T<P1,...,Pn>`. This class models such a `Type` class but ensures that the type is *reified*. Reification means that the Type graph associated with a Java 5 `Type` instance is traversed until the type becomes a concrete class. This class is available with the `getRawClass()` method. The optional type parameters are recursively represented as Reified Types. In Java 1.4, a class has by definition no type parameters. This class implementation provides the Reified Type for Java 1.4 by making the raw class the Java 1.4 class and using a Reified Type based on the `Object` class for any requested type parameter. Implementations can subclass this class and provide access to the generic type parameter graph for conversion. Such a subclass must *reify* the different Java 5 `Type` instances into the reified form. That is, a form where the raw Class is available with its optional type parameters as Reified Types. For example:

```
class Foo {
   Map? extends T>> map = ...;
}
```

The Type graph for map will be quite complex due to the variable `T`

Version:
$Id: 7fff2ea924548b4cb4e275065f0a142f8d8467eb $
Immutable

Constructor Summary	Page
ReifiedType(Class<T> clazz) Create a Reified Type for a raw Java class without any generic type parameters.	80

Method Summary		Page
ReifiedType<?>	**getActualTypeArgument**(int i) Return a type parameter for this type.	81
Class<T>	**getRawClass**() Return the raw class represented by this type.	81
int	**size**() Return the number of type parameters for this type.	81

Constructor Detail

ReifiedType

```
public ReifiedType(Class<T> clazz)
```

Create a Reified Type for a raw Java class without any generic type parameters. Subclasses can provide the optional generic type parameter information. Without subclassing, this instance has no type parameters.

Parameters:
`clazz` - The raw class of the Reified Type.

Method Detail

getRawClass

```
public Class<T> getRawClass()
```

Return the raw class represented by this type. The raw class represents the concrete class that is associated with a type declaration. This class could have been deduced from the generics type parameter graph of the declaration. For example, in the following example:

```
Map<String, ? extends Metadata>
```

In this example, the raw class is the Map interface.

Returns:
The raw class represented by this type.

---

getActualTypeArgument

```
public ReifiedType<?> getActualTypeArgument(int i)
```

Return a type parameter for this type. The type parameter refers to a parameter in a generic type declaration given by the zero-based index `i`. For example, in the following example:

```
Map<String, ? extends Metadata>
```

type parameter 0 is `String`, and type parameter 1 is `Metadata`.

This implementation returns a Reified Type that has `Object` as class. Any object is assignable to Object and therefore no conversion is then necessary. This is compatible with versions of Java language prior to Java 5. This method should be overridden by a subclass that provides access to the generic type parameter information for Java 5 and later.

Parameters:
`i` - The zero-based index of the requested type parameter.
Returns:
The `ReifiedType` for the generic type parameter at the specified index.

---

size

```
public int size()
```

Return the number of type parameters for this type.

This implementation returns `0`. This method should be overridden by a subclass that provides access to the generic type parameter information for Java 5 and later.

Returns:
The number of type parameters for this type.

Package org.osgi.service.formatter

Formatter Package Version 1.0.

See:
[Description]

Interface Summary		Page
AggregateFormatter	The Aggregate Formatter service aggregates the Formatter services present in the service registry.	*83*
Formatter	A Formatter service can format an object to a CharSequence.	*84*

Package org.osgi.service.formatter Description

Formatter Package Version 1.0.

The purpose of this package is to provide access to a formatting service.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.formatter; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.formatter; version="[1.0,1.1)"
```

Interface AggregateFormatter

org.osgi.service.formatter
All Superinterfaces:
Formatter

```
public interface AggregateFormatter
extends Formatter
```

The Aggregate Formatter service aggregates the Formatter services present in the service registry. That is, a Formatter service provides the formatting facility for one or more types, an Aggregate Formatter service provides formatting for any object by using the Formatter services present in the service registry. When called, an Aggregate Formatter must use the Formatter.OSGI_FORMATTER_TYPE property of the Formatter services to determine which service to use. Finding the right formatter must follow the following rules:

1. Exact class
2. Exact implemented interfaces in order of declaration
3. Recursively for super class/super interface
4. for each interface in order of declaration: recursively for each interface

If for any of the previous step multiple services apply, the service ranking rules must be used to determine the correct service. An Aggregate Formatter must not register with a Formatter.OSGI_FORMATTER_TYPE service property. The Aggregate Formatter service uses the same signature as the Formatter Service, the interface is therefore a marker interface to simplify getting access to the aggregator.

ThreadSafe

Fields inherited from interface org.osgi.service.formatter.Formatter
INSPECT, LINE, OSGI_FORMATTER_TYPE, PART

Methods inherited from interface org.osgi.service.formatter.Formatter
format

Interface Formatter

org.osgi.service.formatter
All Known Subinterfaces:
AggregateFormatter

```
public interface Formatter
```

A Formatter service can format an object to a CharSequence. The purpose of a formatter is to create human readable output from general objects. The primary purpose is for the command shell, but other uses do exist. A key concept in the formatter is the level of detail that is displayed. In practice, the detail can vary depending on the use case. For this reason, there are three levels:

1. INSPECT - Provides the greatest level of detail. The output can consist of multiple lines. If columns are used, it is recommended to use 40 characters for the first column. However, the format of the output is more or less free.
2. LINE - This level is used to print collections. The output may consist of multiple lines but columns should have fixed width so they align.
3. PART - A cell in a table.

Multiple lines must be separated with LF character ( ), not a CR or CR/LF. The end of the output must never be ended with a LF. The receiver must properly separate the output. Ranking. If two formatters have the same preference for conversion than the service with the highest ranking must be used. A formatter should follow the Locale objects given in the call. This list is in preferential order. If no locales are given the preference is the default locale and finally English.

Version:
$Id: 731d9ee1654cd17539017dd41793d42de31f7efa $
ThreadSafe

Field Summary		Page
int	**INSPECT** Print the object in detail using as many lines and columns as needed.	85
int	**LINE** Print the object as a row in a table.	85
String	**OSGI_FORMATTER_TYPE** This property is String+.	84
int	**PART** Print the value in a small format so that it is identifiable.	85

Method Summary		Page
String	**format**(Object target, int level, Locale... locales) Convert an object to a CharSequence object in the requested format.	85

Field Detail

OSGI_FORMATTER_TYPE

```
public static final String OSGI_FORMATTER_TYPE = "osgi.formatter.type"
```

This property is String+. A string, or array/collection of strings, and specifies the names of the classes and/or interfaces that this Formatter service recognizes. Not setting this property indicates that this is an *aggregate formatter*. An aggregate formatter promises to use the existing Formatter services to convert and is thus not type specific.

INSPECT

```
public static final int INSPECT = 0
```

Print the object in detail using as many lines and columns as needed. For example, a Bundle object would be formatted with all its details like id, location, all its headers, registered services, etc. For this level, completeness is more important than space. The output can contain multiple lines but should not end in a LF (\n,).

LINE

```
public static final int LINE = 1
```

Print the object as a row in a table. The columns should align for multiple objects printed beneath each other. For example, a bundle would print the primary information like the id, the state, the name, the version, but it would ignore the headers, registered services, etc. The print may run over multiple lines but must not end in a LF (\n,).

PART

```
public static final int PART = 2
```

Print the value in a small format so that it is identifiable. For example, for a bundle it would print the id or bundle-symbolic name combination. If applicable, the constructor of the given class (assuming it is not an interface) should take the result of this formatting to reconstruct this object.

Method Detail

format

```
String format(Object target,
              int level,
              Locale... locales)
       throws 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 `CharSequence` object

`level` - One of INSPECT, LINE, or PART.

`locales` - A list of locales in order of preference. If no locales are specified, use the default locale and then english.

Returns:

A printed object of potentially multiple lines, must never be `null`. The return is a CharSequence because this is usually more efficient.

Throws:

`Exception` - If something fails during the formatting ### Should we make this with a StringBuilder arg?

Package org.osgi.service.threadio

Thread IO Package Version 1.0.

See:
Description

Interface Summary		Page
ThreadIO	Enable multiplexing of the standard IO streams for input, output, and error.	*87*
ThreadIO.Entry		*88*

Package org.osgi.service.threadio Description

Thread IO Package Version 1.0.

The purpose of this package is to provide thread based system IO. This service allows a unique System.out, System.in, and System.err stream on a per thread basis.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.threadio; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.threadio; version="[1.0,1.1)"
```

Interface ThreadIO

org.osgi.service.threadio

```
public interface ThreadIO
```

Enable multiplexing of the standard IO streams for input, output, and error. This service provides access the System I/O streams on a per thread basis. The standard streams are singletons, this service replaces the singletons with a service that has a per-thread stack of streams. If no streams are pushed output is directed to the original streams that were set before this service was started. Users can push a triplet of streams. After this push, all standard IO is redirected through the given streams for that thread only. When the user is ready, he can pop the streams and the previous situation is restored.

Version:
$Id: fad8fafb1cf7ef194cc7e7591d2e84ac271d67c3 $
ThreadSafe

Nested Class Summary		Page
static interface	ThreadIO.Entry	88

Method Summary		Page
ThreadIO.Entry	push(InputStream in, PrintStream out, PrintStream err) Associate these streams with the current thread.	87

Method Detail

push

```
ThreadIO.Entry push(InputStream in,
                    PrintStream out,
                    PrintStream err)
```

Associate these 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. If a `null` is given for any of the parameters the stream must not be replaced. This method can be called multiple times on the same thread. A Thread IO implementation must stack these calls. It is paramount that users of this service ensure they follow the bracketing. The following code snippet provides such a model:

```
Entry e  = threadio.push(in, out, err);
try {
    ... do real work
} finally {
   e.pop();
}
```

The streams will automatically be canceled when the bundle that has gotten this service is stopped or returns this service. If the ThreadIO service holds the only reference to a stream, it must remove this stream from the stack of streams.

Parameters:
`in` - InputStream to use for the current thread when System.in is used. If `null` then ignore.
`out` - PrintStream to use for the current thread when System.out is used. If `null` then ignore.
`err` - PrintStream to use for the current thread when System.err is used. If `null` then ignore.
Returns:
and entry that can be used to close/pop the push for life cycle control.

Interface ThreadIO.Entry

org.osgi.service.threadio
Enclosing class:
ThreadIO

public static interface **ThreadIO.Entry**

Method Summary		Page
void	close()	88

Method Detail

close

void **close**()

6 Alternatives

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.

The specification hould 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

8.1 Author's Address

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

8.2 Acronyms and Abbreviations

8.3 End of Document