Tandem Systems, Ltd.

WinAgents HyperConf Scripting

User's Reference

Tandem Systems, Ltd.
2008

Table of Contents

Overview 3

GetCurrentDevice Function 4

WriteLog Function 5

Device Object 6

Device.Connect method 9

Device.Login method 10

Device.SwitchToPrivilegedMode method 11

Device.Send method 12

Device.Receive method 13

Device.ExecuteCommand method 15

Device.Disconnect method 17

Device.IsConnected method 18

Device.IsLoggedIn method 19

Device.IsInPrivilegedMode method 20

Device.EnableConsoleEcho method 21

Host Object 23

Host.OpenInNewWindow method 24

Host.Echo method 25

Using Device Object 26

Handling Device Output 27

Using Regular Expressions 28

Overview

WinAgents HyperConf allows users to execute scripts across network devices. Two kinds of scripts are supported: command scripts and scripts in Microsoft JScript. Command scripts are sequence of device commands executed directly in a CLI session. Scripts in JScript allow implementing more advanced logic and device output processing. WinAgents HyperConf provides a set of functions and objects that are used to establish communication and execute commands on devices in a network.

WinAgents HyperConf supports all the functions of Microsoft's JScript implementation. In addition, WinAgents HyperConf defines the following functions and objects.

Functions

GetCurrentDevice	Returns a device object for which the script is invoked
WriteLog	Outputs a string into HyperConf's operation log window

Objects

Device	Represents device registration in HyperConf and provides methods to communicate with the network device
Host	Allows the user script to manipulate HyperConf application and perform general tasks

GetCurrentDevice Function

The **GetCurrentDevice** function returns a device object for which the script is invoked.

Syntax

```
Device GetCurrentDevice ()
```

Parameters

This function has no parameters.

Return Value

The return value is a device object.

Remarks

It is usual to invoke this function in the beginning of the script to obtain the device object which was selected to execute script for.

Example

For an example, see Using Device Object.

WriteLog Function

The **WriteLog** function outputs a string into operation log window.

Syntax

```
void WriteLog (outputString)
```

Parameters

outputString

[in] String to be displayed.

Return Value

This function does not return a value.

Remarks

This function uses operation log to output the string. String can contain new-line characters (‘\n’) to represent multi-line messages. All the text written by this function is displayed in gray to distinguish it from the actual device output.

Example

For an example, see Using Device Object.

Device Object

The Device object represents device registration in HyperConf. It contains properties to retrieve device registration information (such as device address, user credentials, etc.) and methods to perform common communication operations between script and device.

Methods

The Device object defines the following methods.

Connect	Establishes a connection to the device
Login	Implements login procedure on the device
SwitchToPrivilegedMode	Enters privileged mode on the device
Send	Sends a string to the device
Receive	Receives data from the device
ExecuteCommand	Executes a command on the device
Disconnect	Closes the connection to the device
IsConnected	Checks whether the connection to the device is established
IsLoggedIn	Checks whether login procedure is performed and completed
IsInPrivilegedMode	Checks whether the device session is in privileged mode
EnableConsoleEcho	Switches on/off device output to HyperConf's operation log

Properties

The Device object has the following properties. These properties represent values configured in HyperConf in the device registration.

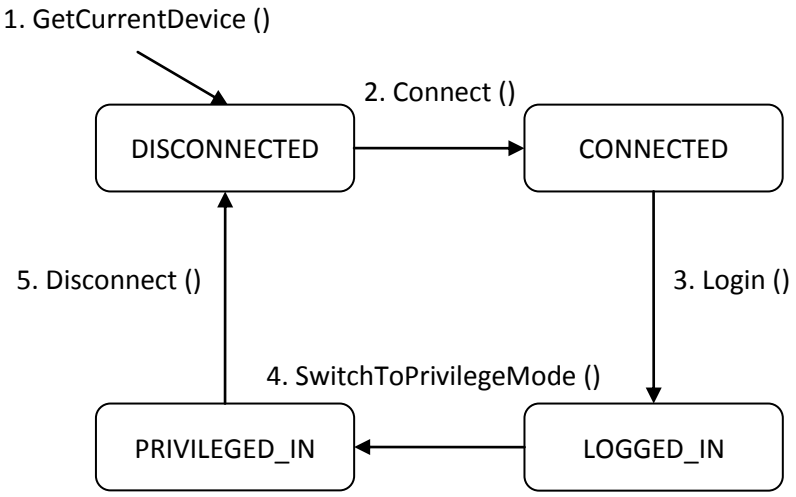
properties.Address	Device address
properties.TelnetPort	Port number for TELNET protocol
properties.SshPort	Port number for SSH protocol
properties.ConsoleEx.ExecPrompt	Exec or non-privileged mode prompt on the device
properties.ConsoleEx.UsernamePrompt	Username prompt on the device
properties.ConsoleEx.PasswordPrompt	Password prompt on the device
credentials.Username	User name used to perform login to the device
credentials.Password	User password used to perform login to the device
credentials.EnablePassword	User password used to enter to privileged mode

template.Vendor	Vendor of the device
template.Model	Model string of the device
template.Text	Device template internal identifier

Device object states

At any given time, the device object can be in one of several states.

DISCONNECTED	Device object is disconnected from the physical device. Methods Login(), SwitchToPrivilegedMode(), Send(), Receive(), ExecuteCommand() cannot be used when the device object is in DISCONNECTED state. This is initial state of the device object after it was obtained via GetCurrentDevice () function or after Disconnect() method was called.
CONNECTED	Device object is connected to the physical device and the terminal protocol is active.
LOGGED_IN	Device object is connected to the physical device and login procedure was successfully completed. The terminal session is active, the device is in USER EXEC mode, it sent command prompt and waits for commands.
PRIVILEGED_IN	Device object is connected to the physical device. The terminal session is active, the device is in PRIVILEGED EXEC mode, it sent command prompt and waits for commands.



1. When script is started there is no device object created and accessible to it. To obtain a device object for which the script was invoked it calls GetCurrentDevice() function. This function creates and initializes the device object and returns it to the calling environment. The script code stores this object in a variable. The device object is in DISCONNECTED state which means

that the terminal protocol is down and the device object is unable to communicate with the physical device.

2. To up the terminal protocol and establish connection to the physical device script calls Connect() method. This method creates a protocol handler and initiates a terminal session with the physical device. All settings for the protocol handler, such as physical device address, port number for the protocol are stored in the device object properties. Upon successful completion Connect() method puts the device object in CONNECTED state. In this state script can call Login(), Send(), Receive(), and ExecuteCommand() methods.
3. To execute login procedure on the device script calls Login() method. This method sends username and password to the physical device and then puts the device object in LOGGED_IN state.
4. To enter the privileged EXEC mode script calls SwitchToPrivilegeMode() methods. This method sends 'enable' or similar command and privileged mode password if needed to the device. Then it puts the device object in PRIVILEGED_IN state.
5. After all actions with the device was performed, script calls Disconnect() method. This method closes the terminal session with the physical device, shuts down the terminal protocol and puts the device object back to DISCONNECTED state.

Example

For an example, see Using Device Object.

Device.Connect method

The method **Connect** establishes a connection to the device using a specified protocol.

Syntax

```
bool Connect ([protocol])
```

Parameters

protocol

[in] Protocol to use for connection, this can be one of the following values:

- TelnetProtocol
- SshProtocol

The parameter is optional, if omitted then the protocol which was selected in 'Select Destinations' window is used.

Return Value

This method returns true if connection was established successfully and false otherwise.

Remarks

This method must be called before other session-related methods. After successful execution the device object is set into CONNECTED state.

Example

For an example, see Using Device Object.

Device.Login method

The method **Login** performs login procedure on the device.

Syntax

```
bool Login ()
```

Parameters

This method has no parameters.

Return Value

This method returns true if login procedure was performed successfully and the device sent a command prompt and is ready to receive commands; and false otherwise.

Remarks

This method can be called after Connect() method to perform login procedure. During login procedure the physical device sends prompts for username and password, and the script replies with username and password that is configured in the Device Registration Window in HyperConf. After successful execution the device object is set into LOGGED_IN state. Your script may perform login procedure manually; see Logging in to the device.

Example

For an example, see Using Device Object.

Device.SwitchToPrivilegedMode method

The method **SwitchToPrivilegedMode** enters PRIVILEGED EXEC mode on the device.

Syntax

```
bool SwitchToPrivilegedMode ()
```

Parameters

This method has no parameters.

Return Value

This method returns true if the terminal session entered privileged mode, the device sent a privileged command prompt and is ready to receive commands; and false otherwise.

Remarks

It is usual to call this method after Login() method to access device capabilities available in the PRIVILEGED EXEC mode. SwitchToPrivilegedMode () method sends to the physical device 'enable' (or similar command), waits for the password prompt and replies with enable password configured in the Device Registration Window in HyperConf. After successful execution the device object is set into PRIVILEGED_IN state. Your script may perform this procedure manually; see Entering PRIVILEGED EXEC mode.

Example

For an example, see Using Device Object.

Device.Send method

The method **Send** sends a string of text to the device via terminal protocol in the established session.

Syntax

```
bool Send (stringToSend)
```

Parameters

stringToSend

[in] String of text to send to the device.

Return Value

This method does not return a value.

Remarks

This method sends the string exactly as it specified by the stringToSend parameter. Terminal session must be established with the physical device, and the device object must be at least in CONNECTED state.

Example

The following code snippet connects to the device, sends 'y' character and then disconnects.

```
// obtain device object
var device = GetCurrentDevice ();

// establish terminal session
if (device.Connect ()) {

   // send 'y' character
   device.Send ("y");

   // and then disconnect
   device.Disconnect ();
}
```

Device.Receive method

The method **Receive** receives an output from the device via terminal protocol.

Syntax

```
string Receive ([waitRegexp1, waitRegexp2, …])
```

Parameters

waitRegexp1, waitRegexp2

> [in] Regular expressions to match the device output against. These parameters are optional. You can specify as many regular expressions as needed.

Return Value

This method returns whole device output received from the device before one of the wait expression would be encountered in the device output or when timeout expires.

Remarks

This method receives the console output from the physical device and stores all the data in an internal buffer. Every time Receive () method receives the data it matches the whole internal buffer against set of wait expressions specified in the parameter list. If the buffer matches one of the wait expressions, method immediately returns the whole data stored in the internal buffer. If no match is found, it continues waiting. If the device doesn't send any data during timeout period, method terminates waiting and returns all the data received. Then the script can analyze the device output, for details see Handling Device Output. For the information about regular expressions, see Using Regular Expressions.

Terminal session must be established with the physical device before calling Receive() method, and the device object must be at least in CONNECTED state.

Example

The following code snippet connects to the device, waits for either "Username: " or "Password: " prompt, and if encounters "Username: " prompt sends username "joe".

```
// obtain device object
var device = GetCurrentDevice ();

// establish terminal session
if (device.Connect ()) {

   // receive device output and search for "Username: "
   var output = device.Receive ("Username: ", "Password: ");
   if (output.match ("Username: ")) {

      // send username "joe" and press ENTER key
```

```
      device.Send ("joe\n");
   }

   // and then disconnect
   device.Disconnect ();
}
```

Device.ExecuteCommand method

The method **ExecuteCommand** executes command on the device via terminal protocol in the established session.

Syntax

```
string ExecuteCommand (command, [waitRegexp1, waitRegexp2, …])
```

Parameters

command

[in] Command string to execute on the device.

waitRegexp1, waitRegexp2

[in] Regular expressions to match the device output against. These parameters are optional. You can specify as many regular expressions as needed.

Return Value

This method returns whole device output received from the device as a result of command execution before device turns command prompt, one of the wait expression would be encountered in the device output, or when timeout expires.

Remarks

This method sends the command string plus new-line character, so it presses ENTER key after command. Then it waits for the device console output, receives and stores it in an internal buffer. Every time the method receives the data it matches the whole internal buffer against set of wait expressions specified in the parameter list. If the buffer matches one of the wait expressions, method immediately returns the whole data stored in the internal buffer. If no match found, it checks whether device returned the command prompt, and if it did, method returns. Otherwise, it continues waiting. If the device doesn't send any data during timeout period, method terminates waiting and returns all the data received. Then the script can analyze device output, for details see Handling Device Output. For the information about regular expressions, see Using Regular Expressions.

Terminal session must be established with the physical device, and the device object must be at least in CONNECTED state.

Example

The following code snippet connects to the device, executes “show version” command and then disconnects.

```
// obtain device object
var device = GetCurrentDevice ();

// establish terminal session
if (device.Connect ()) {

   // perform login procedure
   if (device.Login ()) {

      // execute command
      device.ExecuteCommand ("show version");
   }

   // and then disconnect
   device.Disconnect ();
}
```

Device.Disconnect method

The method **Disconnect** disconnects from the device and shuts down the terminal session.

Syntax

```
void Disconnect ();
```

Parameters

This method has no parameters.

Return Value

This method does not return a value.

Remarks

This method must be called after all actions with the device were performed. After successful execution the device object is set into DISCONNECTED state.

Example

For an example, see Using Device Object.

Device.IsConnected method

The method **IsConnected** checks if the terminal session is established with the device.

Syntax

```
bool IsConnected ();
```

Parameters

This method has no parameters.

Return Value

This method returns true if the terminal session is established with the device and false otherwise.

Remarks

The script may call IsConnected() method to check whether the device object is connected to the physical device and it can perform session-related operations such as sending and receiving data.

Example

The following code snippet ensures that connection to the device was established and then performs login procedure.

```
// if not already connected then connect
if (!device.IsConnected ())
   device.Connect ();

// perform login procedure
device.Login ();
```

Device.IsLoggedIn method

The method **IsLoggedIn** checks if login procedure was performed and the physical device ready to receive commands.

Syntax

```
bool IsLoggedIn ();
```

Parameters

This method has no parameters.

Return Value

This method returns true if the terminal session login procedure was performed successfully and false otherwise.

Remarks

The script may call IsLoggedIn() method to check whether the physical device object is either in USER EXEC or PRIVILEGED EXEC mode and waits for commands.

Example

The following code snippet checks if the device is ready to receive commands and executes “show version” command.

```
// if device is ready
if (device.IsLoggedIn ()) {

   // execute “show version”
   device.ExcuteCommand ("show version");
}
```

Device.IsInPrivilegedMode method

The method **IsInPrivilegedMode** checks if the terminal session is in PRIVILEGED EXEC mode.

Syntax

```
bool IsInPrivilegedMode ();
```

Parameters

This method has no parameters.

Return Value

This method returns true if the terminal session is in PRIVILEGED EXEC mode and false otherwise.

Remarks

The script may call IsInPivilegedMode() method to check whether the physical device object is in PRIVILEGED EXEC mode before performing privileged operation with the device.

Example

The following code snippet checks if the device is in PRIVILEGED EXEC mode and executes "show running-config" command.

```
// if device is in PRIVILEGED mode
if (device.IsInPrivilegedMode ()) {

   // execute "show running-config"
   device.ExecuteCommand ("show running-config ");
}
```

Device.EnableConsoleEcho method

The method **EnableConsoleEcho** switches on/off device output to HyperConf's operation log.

Syntax

```
void EnableConsoleEcho (bool enable);
```

Parameters

enable

[in] If true, then all device output will be written into HyperConf's operation log. If false, device output will not be displayed.

Return Value

This method does not return a value.

Remarks

By default all the data that the physical device sends to the script is automatically written to the HyperConf's operation log. This may be inconvenient when the script performs a lot of background work, and only a small amount of actual device output is needed, so the script can switch off console output, perform background operations, and then switch console output back on to capture the data needed.

Example

The following code snippet connects to the device, switches console output off, so messages for login procedure and entering privileged mode would not be displayed. Then it switched console output on and executes "show version" command.

```
// obtain device object
var device = GetCurrentDevice ();

// establish terminal session
if (device.Connect ()) {

   // switch off console output
   device.EnableConsoleEcho (false);

   // perform login procedure
   if (device.Login ()) {

      // entering privileged mode
      if (device.SwitchToPrivilegedMode ()) {

         // switch console echo back on
         device.EnableConsoleEcho (true);

         // execute command
         device.ExecuteCommand ("show version");
      }
```

```
    }

    // and then disconnect
    device.Disconnect ();
}
```

Host Object

The **Host** object is a global object that allows the user script to manipulate HyperConf application and perform general tasks.

Methods

The Host object defines the following methods.

OpenInNewWindow	Opens a new window in HyperConf with the text editor
Echo	Shows a message box

Host.OpenInNewWindow method

The method **OpenInNewWindow** opens a new window in HyperConf with the text editor.

Syntax

```
void OpenInNewWindow (text, caption);
```

Parameters

text

[in] Text to place to the text editor.

caption

[in] Caption of the opened window.

Return Value

This method does not return a value.

Remarks

This method can be used to open a window in HyperConf and put into it some text received from the device.

Example

The following code snippet downloads the device configuration with "show running-config" command and opens it in a new window.

```
// obtain device object
var device = GetCurrentDevice ();

// establish terminal session
if (device.Connect ()) {

   // perform login procedure
   if (device.Login ()) {

      // entering privileged mode
      if (device.SwitchToPrivilegedMode ()) {

         // execute command
         var config = device.ExecuteCommand ("show running-config");

         // and open it
         Host.OpenInNewWindow (config, "Config from " + device.properties.Address);
      }
   }

   // and then disconnect
   device.Disconnect ();
}
```

Host.Echo method

The method **Echo** shows a message box with OK button.

Syntax

```
void Echo (text);
```

Parameters

text

[in] Message text to show.

Return Value

This method does not return a value.

Remarks

This method shows modal dialog box with the specified message text and pauses script execution until a user presses OK button.

Example

The following code snippet shows address of the device for which script was invoked.

```
// obtain device object
var device = GetCurrentDevice ();

// show device address
Host.Echo ("Script for " + device.properties.Address);
```

Using Device Object

The Device object allows the script to communicate with the physical device via selected terminal protocol. The script is invoked for each device separately, so it is possible to communicate with only one device at a time. If more than one device is selected in the "Select Destinations" window, then the script will be invoked multiple times, one time per each device. Usually the script performs the following sequence of actions:

1. Obtains the device object which represents the device for which the script was invoked
2. Establishes a connection to the device
3. Performs login procedure
4. Switches a session to PRIVILEGED EXEC mode if needed
5. Executes a set of commands

The following code snippet shows this sequence:

```
// obtain device object
var device = GetCurrentDevice ();
WriteLog ("The device is: " + device.properties.Address);

// establish terminal session
if (device.Connect ()) {

   // perform login procedure
   if (device.Login ()) {

      // here we are in the USER EXEC mode and can execute commands
      device.ExecuteCommand ("show version");
   }
   else
      WriteLog ("Error logging in the device");

   // and diconnect from the device in the end
   device.Disconnect ();
}
else
   WriteLog ("Error connecting to the device");
```

Handling Device Output

Methods of the Device object that receive console session output return all the data received as a JScript string. This string can contain multiple lines, delimited by CR '\r' and LF '\n' characters. To analyze this output the script can use built-in JScript's String and RegExp objects.

Splitting multiline device output into array of strings

```
var output = device.ExecuteCommand ("show version");
var strings = output.split (/[\r\n]+/);
// then iterate through the array
for (var i = 0; i < strings.length; ++i) {
   // handle strings[i]
}
```

Searching for "Username:" text in the output:

```
var output = device.Receive ();
if (output.indexOf ("Username: ") != -1) {
   // device sent prompt for the username
}
```

Searching for "Username:" text in the output using regular expressions:

```
var output = device.Receive ();
if (output.match (/Username: /)) {
   // device sent prompt for the username
}
```

Using Regular Expressions

A regular expression is a pattern of text that consists of ordinary characters (for example, letters a through z) and special characters, known as *metacharacters*. The pattern describes one or more strings to match when searching a body of text. The regular expression serves as a template for matching a character pattern to the string being searched.

Here are some examples of regular expression you might encounter:

JScript	**Matches**
/^\s[\t]*$/	Match a blank line.
/\d{2}-\d{5}/	Validate an ID number consisting of 2 digits, a hyphen, and another 5 digits.

The following table contains the complete list of metacharacters and their behavior in the context of regular expressions:

Character	**Description**
\	Marks the next character as a special character, a literal, a backreference, or an octal escape. For example, 'n' matches the character "n". '\n' matches a newline character. The sequence '\\' matches "\" and "\(" matches "(".
^	Matches the position at the beginning of the input string.
$	Matches the position at the end of the input string.
*	Matches the preceding character or subexpression zero or more times. For example, zo* matches "z" and "zoo". * is equivalent to {0,}.
+	Matches the preceding character or subexpression one or more times. For example, 'zo+' matches "zo" and "zoo", but not "z". + is equivalent to {1,}.
?	Matches the preceding character or subexpression zero or one time. For example, "do(es)?" matches the "do" in "do" or "does". ? is equivalent to {0,1}
{n}	n is a nonnegative integer. Matches exactly n times. For example, 'o{2}' does not match the 'o' in "Bob," but matches the two o's in "food".
{n,}	n is a nonnegative integer. Matches at least n times. For example, 'o{2,}' does not match the "o" in "Bob" and matches all the o's in "foooood". 'o{1,}' is equivalent to 'o+'. 'o{0,}' is equivalent to 'o*'.
{n,m}	m and n are nonnegative integers, where n <= m. Matches at least n and at most m times. For example, "o{1,3}" matches the first three o's in "fooooood". 'o{0,1}' is equivalent to 'o?'. Note that you cannot put a space between the comma and the numbers.
?	When this character immediately follows any of the other quantifiers (*, +, ?, {n},

	{n,}, {n,m}), the matching pattern is non-greedy. A non-greedy pattern matches as little of the searched string as possible, whereas the default greedy pattern matches as much of the searched string as possible. For example, in the string "oooo", 'o+?' matches a single "o", while 'o+' matches all 'o's.
.	Matches any single character except "\n". To match any character including the '\n', use a pattern such as '[\s\S].
(pattern)	Matches pattern and captures the match. The captured match can be retrieved from the resulting Matches collection, using the $0…$9 properties in JScript. To match parentheses characters ( ), use '\(' or '\)'.
(?:pattern)	Matches pattern but does not capture the match, that is, it is a non-capturing match that is not stored for possible later use. This is useful for combining parts of a pattern with the "or" character (|). For example, 'industr(?:y|ies) is a more economical expression than 'industry|industries'.
(?=pattern)	Positive lookahead matches the search string at any point where a string matching pattern begins. This is a non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?=95|98|NT|2000)' matches "Windows" in "Windows 2000" but not "Windows" in "Windows 3.1". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.
(?!pattern)	Negative lookahead matches the search string at any point where a string not matching pattern begins. This is a non-capturing match, that is, the match is not captured for possible later use. For example 'Windows (?!95|98|NT|2000)' matches "Windows" in "Windows 3.1" but does not match "Windows" in "Windows 2000". Lookaheads do not consume characters, that is, after a match occurs, the search for the next match begins immediately following the last match, not after the characters that comprised the lookahead.
x|y	Matches either x or y. For example, 'z|food' matches "z" or "food". '(z|f)ood' matches "zood" or "food".
[xyz]	A character set. Matches any one of the enclosed characters. For example, '[abc]' matches the 'a' in "plain".
[^xyz]	A negative character set. Matches any character not enclosed. For example, '[^abc]' matches the 'p' in "plain".
[a-z]	A range of characters. Matches any character in the specified range. For example, '[a-z]' matches any lowercase alphabetic character in the range 'a' through 'z'.
[^a-z]	A negative range characters. Matches any character not in the specified range. For example, '[^a-z]' matches any character not in the range 'a' through 'z'.
\b	Matches a word boundary, that is, the position between a word and a space. For

	example, 'er\b' matches the 'er' in "never" but not the 'er' in "verb".
\B	Matches a nonword boundary. 'er\B' matches the 'er' in "verb" but not the 'er' in "never".
\cx	Matches the control character indicated by x. For example, \cM matches a Control-M or carriage return character. The value of x must be in the range of A-Z or a-z. If not, c is assumed to be a literal 'c' character.
\d	Matches a digit character. Equivalent to [0-9].
\D	Matches a nondigit character. Equivalent to [^0-9].
\f	Matches a form-feed character. Equivalent to \x0c and \cL.
\n	Matches a newline character. Equivalent to \x0a and \cJ.
\r	Matches a carriage return character. Equivalent to \x0d and \cM.
\s	Matches any white space character including space, tab, form-feed, and so on. Equivalent to [\f\n\r\t\v].
\S	Matches any non-white space character. Equivalent to [^ \f\n\r\t\v].
\t	Matches a tab character. Equivalent to \x09 and \cI.
\v	Matches a vertical tab character. Equivalent to \x0b and \cK.
\w	Matches any word character including underscore. Equivalent to '[A-Za-z0-9_]'.
\W	Matches any nonword character. Equivalent to '[^A-Za-z0-9_]'.
\xn	Matches n, where n is a hexadecimal escape value. Hexadecimal escape values must be exactly two digits long. For example, '\x41' matches "A". '\x041' is equivalent to '\x04' & "1". Allows ASCII codes to be used in regular expressions.
\num	Matches num, where num is a positive integer. A reference back to captured matches. For example, '(.)\1' matches two consecutive identical characters.
\n	Identifies either an octal escape value or a backreference. If \n is preceded by at least n captured subexpressions, n is a backreference. Otherwise, n is an octal escape value if n is an octal digit (0-7).
\nm	Identifies either an octal escape value or a backreference. If \nm is preceded by at least nm captured subexpressions, nm is a backreference. If \nm is preceded by at least n captures, n is a backreference followed by literal m. If neither of the preceding conditions exist, \nm matches octal escape value nm when n and m are octal digits (0-7).
\nml	Matches octal escape value nml when n is an octal digit (0-3) and m and l are octal digits (0-7).
\un	Matches n, where n is a Unicode character expressed as four hexadecimal digits. For example, \u00A9 matches the copyright symbol (©).