Creating Typinator™ Scripts

Version 2.0, December 2021

for Typinator 8.11 or newer

Contents

About this document

Starting with version 5.0, Typinator has the ability to execute external scripts and include the result of the script execution in expansions. Using such scripts is fairly easy. When you edit an expansion, just open the {…} pop-up menu next to the expansion field and select the desired script. Typinator inserts a special marker that contains a script invocation, which will be executed when the expansion is about to be inserted in a document.

This document is meant for those who wish to create new scripts. It describes the mechanism and explains how scripts should be written to use the full potential of Typinator's scripting feature. It assumes that you are already familiar with programming in AppleScript, JavaScript, shell scripting, Perl, PHP, Python, Ruby, or with the development of command line tools in C or Objective C.

This document is accompanied with a few samples that you can install and modify. Some of these samples are mentioned in the text, so we recommend that you put the "Samples" and "Hello" folders into the "Includes" folder.

Starting with macOS Monterey (12.0), PHP is no longer part of a default installation of macOS. The current version 2.0 of this document describes the changes made in Typinator 8.11 to find script interpreters and install PHP.

There is also a section about using Swift as a scripting language. To use Swift, Xcode Tools 7 or newer must be installed.

Integration of scripts in Typinator

Typinator uses special placeholders of the form {xy} in expansions. These placeholders get replaced with dynamic text, which is determined at the time when the expansion takes place. For example, {h24}:{m} gets replaced with the hour and minute of the current time. Typinator uses the same notation and technique for scripts. A script invocation has the form:

    {folder/scriptName argument}

where the argument is optional. When the first part (folder/scriptName) refers to an executable script inside the Includes folder, Typinator invokes the script with the given parameter and replaces the entire placeholder with the result of the script invocation.

Scripts are stored in the Includes folder, which in turn resides in Typinator's sets folder. To quickly open the Includes folder, use the command at the bottom of the {…} menu. We recommend to store scripts not at the top level of the Includes folder, but rather inside a nested folder. Nested folders appear as submenus inside the {…} menu and make it easier to organize scripts by topics.

AppleScript files

To create an AppleScript file for use with Typinator, we recommend that you use Script Editor and save the script as a text file in a subfolder within the Includes folder.

You can save AppleScript scripts in either "Text" file format (with the extension ".applescript") or as compiled scripts ("Script" file format, with the extension ".scpt"). Starting with version 6.7 of Typinator, we recommend the "scpt" format. Scripts saved in this way are precompiled and therefore load and execute faster when an expansion takes place. If you need to support older Typinator versions, save the scripts in the "applescript" format.

Typinator strips off the extension in the {…} menu, but it includes the extension in the generated placeholder. For example, the script file "IP.applescript", stored in the "Scripts" subfolder, appears as "IP" in the menu and is inserted as {Scripts/IP.applescript}.

JavaScript files

In Typinator 6.7 or newer, and OS X 10.10 (Yosemite) or newer, you can use JavaScript as an alternative to AppleScript. JavaScript files must be saved in the compiled "scpt" file format. If you save a JavaScript as plain text, the compiler incorrectly assumes that the script is written in the AppleScript language.

Simple scripts

When Typinator processes an expansion with a script placeholder, it runs the script and replaces the placeholder with the result of the script. The script therefore needs to return the desired result. If the result is not text, Typinator tries to coerce it to text.

A very simple script could look like this:

```
return "Sample Text"
```

A more meaningful script for inserting the computer's IP number is:

```
return IPv4 address of (system info)
```

In JavaScript, you can simply write a series of statements; the value produced by the last statement is returned as the result:

```
var theNumber = 42
"fourty-two = " + theNumber
```

This script returns the result "fourty-two = 42"

Scripts with parameters

Typinator can invoke scripts with parameters. Parameters are simple strings that follow the script name in the placeholder. To create a script that accepts parameters, it must contain a handler with the name *expand* and at least one parameter. For example, the following script takes a string parameter and returns the length of the string:

```
on expand(str)
    return length of str
end expand
```

If the script has the name *Length*, you can invoke it as

```
{Scripts/Length.applescript parameter}
```

You can even nest script invocations. For example,

```
{Scripts/Length.applescript {Scripts/IP.applescript}}
```

returns the number of characters of the current IP address.

When you create a string with a parameter, we recommend that you include a hint about the meaning of the parameter in the script. You can do this by means of a comment on the same line as the header of the *expand* handler. The comment must begin with “parameter:”:

```
on expand(str) -- parameter: any string
    return length of str
end expand
```

When you select the *Length* script from the menu, Typinator includes the hint as a default parameter in the placeholder:

```
{Scripts/Length.applescript any string}
```

When a script expects a keyword parameter that controls the behavior of the script, list the possible alternatives, separated by vertical bars, as shown in the *UserName* example:

```
on expand(kind) -- parameter: short|long|computer
    if kind = "short" then
        return (short user name of (system info))
    else if kind = "computer" then
        return (computer name of (system info))
    else -- "long" or anything else
        return (long user name of (system info))
    end if
end expand
```

The corresponding placeholder now looks like this:

```
{Samples/UserName.applescript short|long|computer}
```

This makes it pretty clear what the script expects. Using arguments in this way makes it easy to create a single script for multiple related purposes.

Note that Typinator supports only a single parameter in script invocations. If your script needs multiple parameters, you need to split the parameter into parts yourself. Here is a sample code fragment that splits a single parameter theParameter into an array of parameters, using a space character as a separator.

```
set originalDelimiters to AppleScript's text item delimiters
set AppleScript's text item delimiters to " "
set parameterList to every text item of theParameter
set AppleScript's text item delimiters to originalDelimiters
```

When you use such a technique, also supply a default parameter that makes clear what the script expects.

In JavaScript, a script with a parameter looks like this:

```
function expand(parameter) { //-- parameter: any text
    return "[" + parameter + "]"
}
```

This script encloses the parameter in brackets. Note that the comment with the parameter description must contain two dashes after //, and there must be a single space before the word "parameter".

Testing scripts

You can test your scripts independently of Typinator in Script Editor. Simply run your script and check the result in the Script Editor window. If your script has an expand handler, just include an invocation of the handler in the script. For example, you could test the UserName script as follows:

```
return expand("long") -- for testing in the Script Editor

on expand(kind) -- parameter: short|long|computer
    if kind = "short" then
        return (short user name of (system info))
    else if kind = "computer" then
        return (computer name of (system info))
    else -- "long" or anything else
        return (long user name of (system info))
    end if
end expand
```

In JavaScript, you can test a script similarly: Just add an invocation of the *expand* function at the end of the script file:

```
function expand(parameter) { //-- parameter: any text
    return "[" + parameter + "]"
}
expand("abc")
```

Other scripting languages

AppleScript and JavaScript are special because they are supported natively in macOS. But Typinator can actually run any script that you could invoke from the command line in Terminal. This includes Bash, Perl, PHP, Python, and Ruby.

Files containing scripts developed in these languages do not need to end with a special extension, but they must have the *executable* attribute set. To set this attribute, use the Unix command

chmod +x *filename*

in the Terminal. You will typically also use a Unix shell like the Terminal application to test your scripts.

To specify the scripting language, the script must begin with a “shebang” line that indicates where the script interpreter can be found. This could be a hard-wired path. For example, to run a Python script, the first line might look like this:

```
#!/usr/bin/python
```

For more flexibility, we recommend using the *env* command to run scripts:

```
#!/usr/bin/env python
```

This invocation looks for Python in the paths specified in the PATH environment variable. It is therefore recommended because it is more flexible.

As in AppleScript and JavaScript, you can include a comment that contains “-- parameter:” to give Typinator a hint about expected arguments. In most scripting languages, comments start with a hash (#) character, so a typical parameter comment would look like this:

```
#-- parameter: name
```

Note that a single space character must be present between “--” and “parameter”.

Your scripts can do almost anything; to define the resulting expansion, just write the desired result to standard output. Typinator will interpret the output as UTF-8.

The “Hello” folder that comes with this manual contains a few scripts in different scripting languages. All these scripts do the same thing: They just display the message “Hello, World!”. If invoked with a parameter, they display the parameter in place of the word “World”. Take a look at these examples to see how to process parameters and return a result from scripts. Here is the contents of the Bash sample:

```
#!/usr/bin/env bash
#-- parameter: name
if [ "$1" != "" ]; then
    echo -n "Hello, $1!" # -n suppresses newline
else
    echo -n "Hello, World!"
fi
```

To include this script in an expansion, you would, for example, write

```
HAL said "{Hello/HelloBash Dave}".
```

to produce the following expansion:

```
HAL said "Hello, Dave!”.
```

Starting with macOS 12.0 (Monterey), PHP is no longer included in a default installation of macOS. Typinator 8.11 and newer can detect whether PHP is available; if not, it displays a warning and tells you how you can install and use PHP. You can find the instructions on this web page:

https://www.ergonis.com/products/tips/install-php-on-macos.php

Swift scripts

If you have Xcode Tools 7 or newer installed, you can even use Swift for writing Typinator scripts. The first version of Swift was introduced already with Xcode 6 by Apple, but some significant changes were made to Swift 2 in Xcode 7. The instructions given in this document therefore are not backwards compatible with older Xcode versions.

You can write Swift scripts in the same way as described in the previous section on “Other scripting languages”. The script file must have the executable attribute set, you need to include a shebang line, you can include a parameter description in a comment, and the script produces the result by printing strings to the console. The Swift version of the “Hello” script looks like this:

```
#!/usr/bin/env xcrun swift
//-- parameter: name
var argument = "World"
if Process.arguments.count > 1 {
    argument = Process.arguments[1]
}
print("Hello \(argument)!", terminator:"") // empty terminator suppresses newline
```

The command line arguments are available in Process.arguments. The first argument is the path to the script; the argument supplied by Typinator is in Process.arguments[1]. It is necessary to check the argument count first to avoid an index error.

To suppress a line break at the end (which is the default behavior in Swift 2), you need to specify an empty terminator in the print function.

When writing Swift scripts, you should take extra care to catch all exceptions (not shown in the above example for brevity). If a Swift script crashes because of an error, it returns a stack trace to Typinator, which may be confusing for users.

Compiled command-line tools

As yet another possibility, you can create precompiled command-line tools, written in C or Objective C. The “HelloTool” example is a simple C tool with the following source text:

```
#include <stdio.h>
const char * parameter = "\n-- parameter: name\n";
int main (int argc, const char * argv[]) {
 if (argc > 1) {
    printf("Hello, %s!", argv[1]);
 } else {
    printf("Hello, World!");
 }
 return 0;
}
```

To create an executable command-line tool, you would typically use Xcode and create a “Command Line Tool” project.

Since the resulting tool contains only executable code, you must use a trick to specify the expected parameter: Simply declare a string constant that contains “-- parameter:” and ends with a newline character. The compiler will then include the parameter description in the executable file, and Typinator will be able to extract it.

Important rules

When writing scripts, you should consider that Typinator invokes them in a low-level situation while the user is typing text. Scripts therefore should **execute quickly**, typically in less than a second. For command-line tools and scripting languages other than AppleScript, Typinator limits the available time to 5 seconds. If your script takes longer, Typinator terminates the script without a warning.

This also means that scripts **should not interact with the user**. Do not use any alerts or dialogs inside your scripts.

It is also a good idea to check for errors. If your script might fail, use an exception handler, if possible. In AppleScript, wrap the critical parts in a try clause and return an error message in the error handler.

Environment variables

Typinator sets a few environment variables to tell scripts about the current context. These are:

- *Typinator_version*: The version number of Typinator.
- *Typinator_setsFolder*: The path to the Sets folder, as specified in Typinator’s preferences.
- *Typinator_abbreviation**: The abbreviation that triggered the expansion.
- *Typinator_wholeWord*: The “whole word” setting of the abbreviation (0 or 1).
- *Typinator_formatted*: 1 for formatted text expansion, 0 for plain text.
- *Typinator_HTML*: 1 for expansions of the “HTML” type, 0 for other types. HTML expansions were introduced with Typinator 6.0; this attribute is therefore not available for older versions of Typinator. See the *ContextAS.applescript* sample for how to safely handle this case.
- *Typinator_language*: The language used for date elements in the expansion (such as “en” for English)
- *Typinator_set**: The name of the set that contains the abbreviation.
- *Typinator_matchingText**: The entered text that matches the abbreviation. This may differ from Typinator_abbreviation in case. For example, “Jgv” matches “jgv” when the abbreviation is defined with the option “Case does not matter”.
- *Typinator_precedingText**: The text that was entered immediately before the abbreviation. Note that this is only the typed text that Typinator knows about. When the user clicks in the middle of a paragraph and then types an abbreviation, Typinator does not know anything about the text before the insertion point. Typinator_precedingText will then be the empty string.

- *Typinator_date*: The current date (considering the yearDelta, monthDelta and dayDelta settings) in the form year-month-date, as it would be inserted with the marker combination {YYYY}-{MM}-{DD}.
- *Typinator_application**: The name of the current application in which the expansion will be inserted.
- *Typinator_applicationPath*: The full path of the application.
  (such as "/Applications/TextEdit.app")
- *Typinator_applicationIdentifier*: The unique identifier of the application.
  (such as "com.apple.TextEdit").
- *Typinator_applicationVersion*: The application's version number.

* Environment variables are stored in an 8-bit encoding (MacRoman for AppleScript, UTF-8 for shell scripts). Special characters in text (such as accented letters) may get lost during the translation into 8-bit format.

You can access these variables like any other environment variables. For example, prefix the name with a dollar sign in Bash:

```
#!/usr/bin/env bash
echo "Typinator version: $Typinator_version"
```

In AppleScript, you could use "system attribute" to access the variables:

```
set TypiVersion to system attribute "Typinator_version"
```

However, there is a simpler, more efficient and more robust technique, when Typinator 5.2 or newer is installed. You can then use an expand handler with two parameters, like this:

```
on expand(str, env)
    return Typinator_version of env
end expand
```

The second parameter is an AppleScript record that contains all the environment variables listed above. The advantage of this technique is that text items in this record are in native Unicode format, whereas the "system attribute" technique is limited to the MacRoman character set. The two-parameter version is therefore recommended for all newly developed scripts. However, the documentation of your script should mention that it requires Typinator 5.2 or newer.

The same technique also works for JavaScript. Just add a second parameter for the environment. You can then access the components with indexes:

```
function expand(str, env) {
    return env["Typinator_version"]
}
```

For examples in AppleScript and Bash that list Typinator's special environment variables, see the ContextAS and ContextBash scripts in the Samples folder.

Typinator uses the same mechanism for all variables that are defined in expansions. For example, {{amount=100}} assigns the string "100" to the variable "amount", and you can use {{amount=?Amount}} to make Typinator interactively ask for the Amount. You can then access this variable with

```
set amt to amount of env
```

However, this will raise an exception when the variable "amount" is not (yet) defined. If you are not sure whether or not a variable exists, you should therefore use an exception handler:

```
try
    set amt to amount of env
on error
    set amt to 0
end try
```

Typinator uses the same mechanism also for date and time offsets. For example, when you insert a marker for a date calculation that sets the date to tomorrow, Typinator uses the notation {{dayDelta=+1}}. This creates an environment variable "dayDelta" with the value "+1". Typinator uses these values internally for date markers, but also defines them as environment variables, so scripts can access these values. The following example shows how you can find out whether there is a day offset:

```
try
    set dd to (dayDelta of env) as integer
    if dd = 0 then return "today"
    if dd = 1 then return "tomorrow"
    if dd = -1 then return "yesterday"
    return "some other day"
on error -- dayDelta is undefined; treat this case like dd = 0
    return "today"
end try
```

Note that the environment variables are represented by strings. To use the numeric value, you need to coerce the value to type *integer*.

The notation with the double braces is actually more general than that. It allows you to define *any* variables. For example, you could include something like {{magicNumber=42}} or {{one=1;two=2}} or {{server=www.ergonis.com}} in an expansion. These assignments are processed left-to-right and are preserved across expansions and even restarts.

You can use such assignments to implement states that you can modify with simple abbreviations. For example, you can create two abbreviations *offerOn* and *offerOff* that do not actually expand into anything but just turn a state variable on and off: {{specialOffer=1}} and {{specialOffer=0}}. You can then use these abbreviations to control the behavior of scripts that produce different results depending on the *specialOffer* variable.

Note that the {{…}} assignments should not contain any spaces. When you write {{v=12}} or {{x=first;y=second;}}, Typinator treats everything between the equals sign and the closing braces or semicolon as the string to be assigned. When you write {{v= 12}}, the string " 12" (digits 1 and 2 with a leading space character) is assigned to v.

Note: To avoid conflicts with common Unix environment variables (such as HOME, PATH, SHELL, TMPDIR, and USER), variables used in Typinator must contain at least one lowercase character. Typinator ignores all variables whose names consist only of capital letters and "_".

Returning values to Typinator

You can use the assignment notation to return variables to Typinator. Just include a {{…}} block at the beginning of the returned result and embed assignments between the braces. Typinator removes the {{…}} block and handles the assignments inside. As mentioned above, the variables' names must contain at least one lower-case letter.

Typinator stores the returned variables in a global dictionary and keeps them around in its preferences. This means that scripts can use this mechanism to permanently store values and use them during a subsequent invocation. For an example, see the UsageDays script:

```
on expand(str, env) -- str is not used, but required
    set today to current date
    set todayString to date string of today

    try
        set lastDate to lastDate of env
        set usageDays to (usageDays of env) as integer
    on error
        set lastDate to ""
        set usageDays to 0
    end try

    set assignments to ""
    if lastDate ≠ todayString then -- new day; count the day and update lastDate
        set usageDays to usageDays + 1
        set assignments to "{{lastDate=" & todayString & ";usageDays="
            & usageDays & "}}"
    end if

    return assignments & usageDays
end expand
```

You can now embed this script in an expansion like this:

```
used on {Samples/UsageDays.applescript} days
```

The script maintains two variables: *usageDays* is the number of days on which this script has been used, where multiple uses on the same day count only once. *lastDate* is the date of the last usage. When the current date is the same as the last usage date, the script simply returns the *usageDays* variable. On a new day, it prepares a string that sets the new date and the new usage count. For example, the first invocation on April 26 would return a string like:

```
{{lastDate=Fri 26. April 2014;usageDays=6}}6
```

Typinator processes and removes the assignments. The actual result is then the number 6 right after the closing braces, so that the expansion yields:

```
used on 6 days
```

When used again on the same day, the script leaves the *assignments* variable empty and just returns the previous value of *usageDays*.

You can even include the variables in other expansions without running any script by simply writing the variable name inside a pair of double braces: {{lastDate}}. This placeholder gets replaced with the variable's value (here: the date of the last invocation of the *UsageDays* script).

You can also use this mechanism to create date and time calculations. For example, you can use a script that returns {{yearDelta=0;monthDelta=0;dayDelta=+7}} to set the date one week ahead of the current date.

Calculator variables

The calculator in the Quick Search field also has support for variables. For example, you can enter a calculation like

```
total=120.5+67.3+45
```

When you insert the result in a document, you get 232.8, and Typinator also assigns this value to a variable named *total*. To add 25%, you can later enter

```
total*1.25
```

in the QuickSearch field, which results in the value 291.

These variables are available as environment variables with the prefix "calculator_". In AppleScript, the variable *total* can be accessed as

```
set total to system attribute "calculator_total"
```

or

```
set total to calculator_total of env
```

or in JavaScript:

```
total = env["calculator_total"]
```

To modify the value of a calculator variable, include an assignment in a {{…}} block, as described in the previous section. For example, the following statements define a new variable *amount* that is computed as *total* plus 25%:

```
set amount to total * 1.25
return "{{calculator_amount=" & amount & "}}"
```

After execution of this script, you can use the variable *amount* in the calculator.

You can also use the {{…}} notation to include a calculator variable in an expansion without running a script: {{calculator_amount}}.

Simulating backspaces

Typinator uses a special variable named *bs* to trigger a series of backspace keystrokes before inserting an expansion. In combination with the *Typinator_precedingText* environment variable, you can use this trick for a few cool effects. For example, you can create a script that looks for a number immediately to the left of the typed abbreviation. The script can use this number in its computation of the result, and it can return the number of desired backspaces in the *bs* variable to make Typinator "eat up" the number before inserting the result. For an example of this technique, see the *Repeat* script. This script takes a string as a parameter and repeats this string as often as specified by the number before the typed abbreviation. You can then create an abbreviation "*-" with the following expansion:

```
{Samples/Repeat.applescript -}
```

When you now type "80*-", Typinator replaces this with a series of 80 dashes:

```
--------------------------------------------------------------------------------
```

Here is another example: Create an abbreviation "*paste" with the expansion:

```
{Samples/Repeat.applescript {clip}}
```

In this case, Typinator passes the contents of the clipboard as a parameter to the script, which then duplicates that as many times as you specify when you type the abbreviation. To see the script in effect, copy the text "many, " to the clipboard, and then type "12*paste", and you will get:

```
many, many, many, many, many, many, many, many, many, many, many, many,
```

The Multiply script uses a similar trick. It multiplies the number entered before the abbreviation with the number given as the script's parameter. For example, try an abbreviation "mm with the following expansion:

```
{Samples/Multiply.applescript 25.4}
```

Then enter, for example, 4.5"mm to quickly convert 4.5 inch to mm (11.43).

Note: Repeat and Multiply sample scripts are included just for demonstration purposes. Typinator has a few built-in features that let you do such tricks more elegantly, without the need for scripts. For example, the following regular expression rule also performs the inch-to-mm conversion:

```
regular expression: (\d+\.?\d+)"mm
expansion:          {{#$1*25.4}}
```

The regular pattern matches a sequence of digits with an optional decimal point, followed by a double quote and "mm". The expansion then uses a calculation field to multiply the value of the number ($1) with 25.4. In a similar way, you can use regular expressions in combination with the built-in /Repeat function to create repetitions of entered text.

Canceling expansions

When a script cannot create a meaningful result, you may want to tell Typinator that it should ignore the expansion altogether. To do this, you can use the special variable named *cancelExpansion*:

```
{{cancelExpansion=yes}}
```

The actual value does not matter. The presence of the variable is sufficient to tell Typinator that the expansion should not take place.

Canceling an expansion may be desirable for scripts that use the *Typinator_precedingText* environment variable to look at the text that was typed immediately before the current abbreviation. If this text does not contain any useful information, canceling the expansion may be a good idea. Typinator will then leave the typed abbreviation and the preceding text untouched.

Contributions are welcome

We make useful scripts available for download on our “Download Extras” web page. If you have developed a cool script that you wish to share with other Typinator users, please send it to <typinator-support@ergonis.com>.