Что такое Windows PowerShell и с чем его едят? Часть 2: введение в язык программирования

* [Блог компании RUVDS.com](https://habr.com/ru/company/ruvds/),
* [Системное администрирование](https://habr.com/ru/hub/sys_admin/),
* [Системное программирование](https://habr.com/ru/hub/system_programming/),
* [PowerShell](https://habr.com/ru/hub/powershell/),
* [Разработка под Windows](https://habr.com/ru/hub/win_dev/)

*Исторически утилиты командной строки в Unix-системах развиты лучше чем в Windows, однако с появлением нового решения ситуация изменилась.*  
  
[](https://habr.com/ru/company/ruvds/blog/487876/)  
  
Для PowerShell можно писать сценарии на интерпретируемом мультипарадигменном языке, в котором есть элементы классического процедурного, объектно-ориентированного и даже функционального программирования: условный переход, циклы, переменные, массивы, хэш-таблицы, классы, обработка ошибок, а также функции, командлеты и конвейеры. [**Предыдущая статья**](https://habr.com/ru/company/ruvds/blog/487876/) была посвящена основам работы в среде, а сейчас мы предлагаем вниманию читателей небольшой справочник для программистов.

Оглавление:

[Комментарии](https://habr.com/ru/company/ruvds/blog/490924/#section1)  
[Переменные и их типы](https://habr.com/ru/company/ruvds/blog/490924/#section2)  
[Системные переменные](https://habr.com/ru/company/ruvds/blog/490924/#section3)  
[Области видимости](https://habr.com/ru/company/ruvds/blog/490924/#section4)  
[Переменные окружения (среды)](https://habr.com/ru/company/ruvds/blog/490924/#section5)  
[Арифметические операторы и операторы сравнения](https://habr.com/ru/company/ruvds/blog/490924/#section6)  
[Операторы присваивания](https://habr.com/ru/company/ruvds/blog/490924/#section7)  
[Логические операторы](https://habr.com/ru/company/ruvds/blog/490924/#section8)  
[Условный переход](https://habr.com/ru/company/ruvds/blog/490924/#section9)  
[Циклы](https://habr.com/ru/company/ruvds/blog/490924/#section10)  
[Массивы](https://habr.com/ru/company/ruvds/blog/490924/#section11)  
[Хэш-таблицы](https://habr.com/ru/company/ruvds/blog/490924/#section12)  
[Функции](https://habr.com/ru/company/ruvds/blog/490924/#section13)  
[Обработка ошибок](https://habr.com/ru/company/ruvds/blog/490924/#section14)  
  
Писать код можно в любом текстовом редакторе или с использованием интегрированной среды разработки — проще всего взять Windows PowerShell ISE из комплекта поставки серверных операционных систем Microsoft. Нужно это только для достаточно сложных скриптов: короткие наборы команд проще выполнять в интерактивном режиме.

Комментарии

Использование комментариев считается частью хорошего стиля программирования наряду с правильными отступами и пробелами:

*# Для строчных комментариев используется символ решетки — содержимое строки интерпретатор не обрабатывает.*

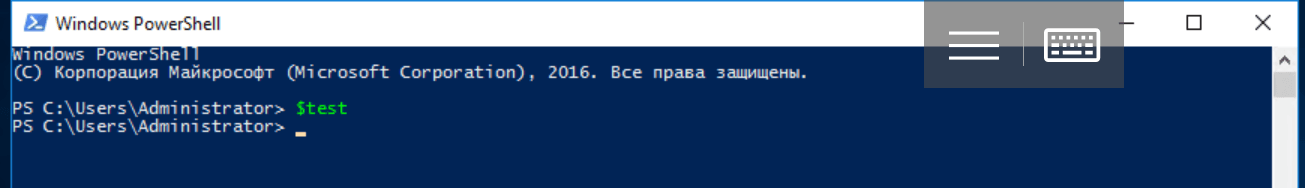
*<#*

*Так обозначаются начало и конец блочного комментария.*

*Заключенный между ними текст интерпретатор игнорирует.*

*#>*

Переменные и их типы

Переменные в PowerShell — это именованные объекты. Их названия могут включать символ подчеркивания, а также буквы и числа. Перед именем всегда используется символ $, а чтобы объявить переменную, достаточно указать интерпретатору допустимое имя:  
  
  
  
Для инициализации переменной (присвоения ей значения) применяется оператор присваивания (символ =):

$test = 100

Объявить переменную можно с указанием ее типа в квадратных скобках (оператор приведения типов) перед именем или значением:

[int]$test = 100

$test = [int]100

Важно понимать, что переменные в PowerShell — это полноценные объекты (классы) со свойствами и методами, типы которых основаны на имеющихся в .NET Core. Перечислим основные:

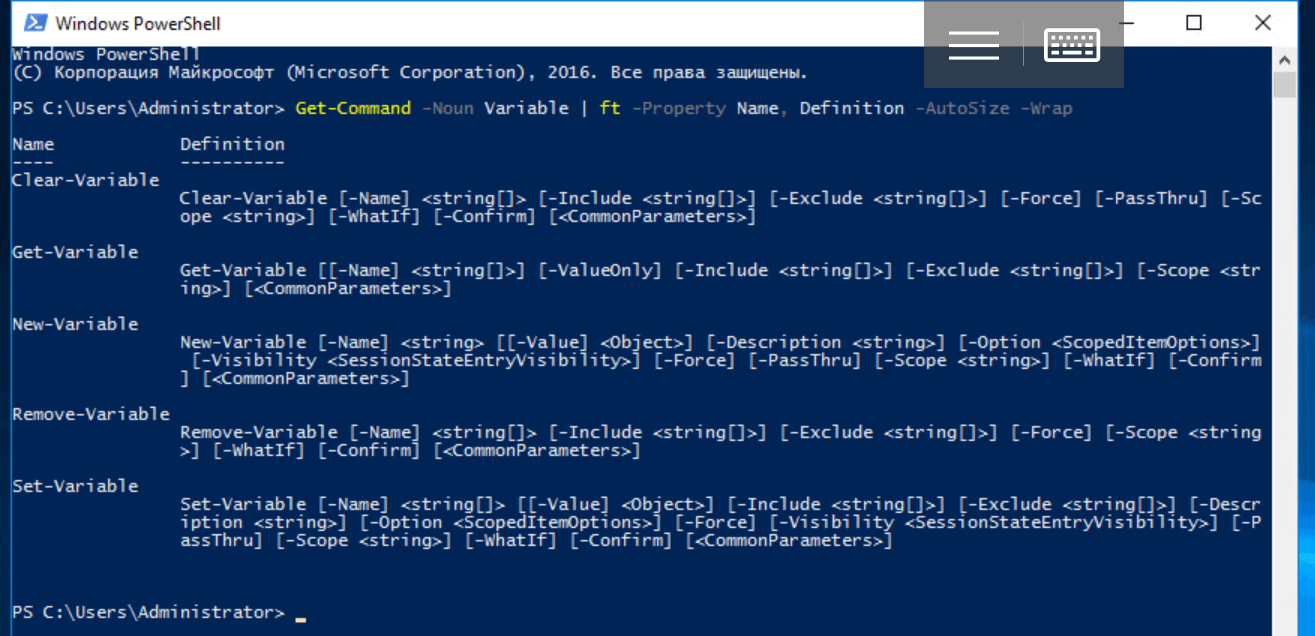
|  |  |  |
| --- | --- | --- |
| **Тип (класс .NET)** | **Описание** | **Пример кода** |
| [string] System.String | строка Unicode | $test = «тест» $test = 'тест' |
| [char] System.Char | символ Unicode (16 бит) | [char]$test = 'c' |
| [bool] System.Boolean | булевский тип (логическое значение True или False) | [bool]$test = $true |
| [int] System.Int32 | тридцатидвухразрядное целое число (32 бита) | [int]$test = 123456789 |
| [long] System.Int64 | шестидесятичетырехразрядное целое число (64 бита) | [long]$test = 12345678910 |
| [single] System.Single | число с плавающей точкой длиною в 32 бита | [single]$test = 12345.6789 |
| [double] System.Double | число с плавающей точкой длиною в 64 бита (8 байт) | [double]$test = 123456789.101112 |
| [decimal] System.Decimal | число с плавающей точкой длиною в 128 бит (обязательно указывать d на конце) | [decimal]$test = 12345.6789d |
| [DateTime] System.DateTime | дата и время | $test = Get-Date |
| [array] System.Object[] | массив, индекс элементов которого начинается с 0 | $test\_array = 1, 2, «тест», 3, 4 |
| [hashtable] System.Collections.Hashtable | хэш-таблицы — ассоциативные массивы с именованными ключами, построенные по принципу: @{ключ = «значение»} | $test\_hashtable = @{one=«один»; two=«два»; three=«три»} |

PowerShell поддерживает неявное преобразование типов, кроме того тип переменной может меняться на ходу (например, при помощи оператора присваивания), если он не указан принудительно — в этом случае интерпретатор выдаст ошибку. Определить тип переменной из предыдущего примера можно при помощи вызова метода GetType():

$test.GetType().FullName

  
  
Существует некоторое количество командлетов для управления переменными. Их список в удобной форме выводится с помощью команды:

Get-Command -Noun Variable | ft -Property Name, Definition -AutoSize -Wrap

  
  
Для просмотра объявленных переменных и их значений можно использовать специальный командлет:

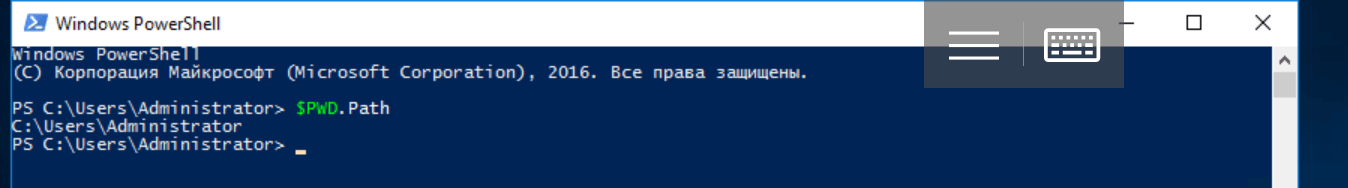
Get-Variable | more

Такой способ кажется чрезмерно громоздким, с переменными намного удобнее работать через операторы или обращаясь к их свойствам и методам напрямую. Тем не менее командлеты имеют право на существование, поскольку позволяют задать некоторые дополнительные параметры. Важно понимать, что пользовательские переменные определены только в рамках текущего сеанса. После закрытия консоли или завершения сценария они удаляются.

Системные переменные

Помимо объявленных пользователем существуют встроенные (системные) переменные, которые не удаляются после завершения текущего сеанса. Делятся они на два типа, при этом данные о состоянии PowerShell хранятся в автоматических переменных, которым нельзя самостоятельно присвоить произвольные значения. К их числу относится, например, $PWD:

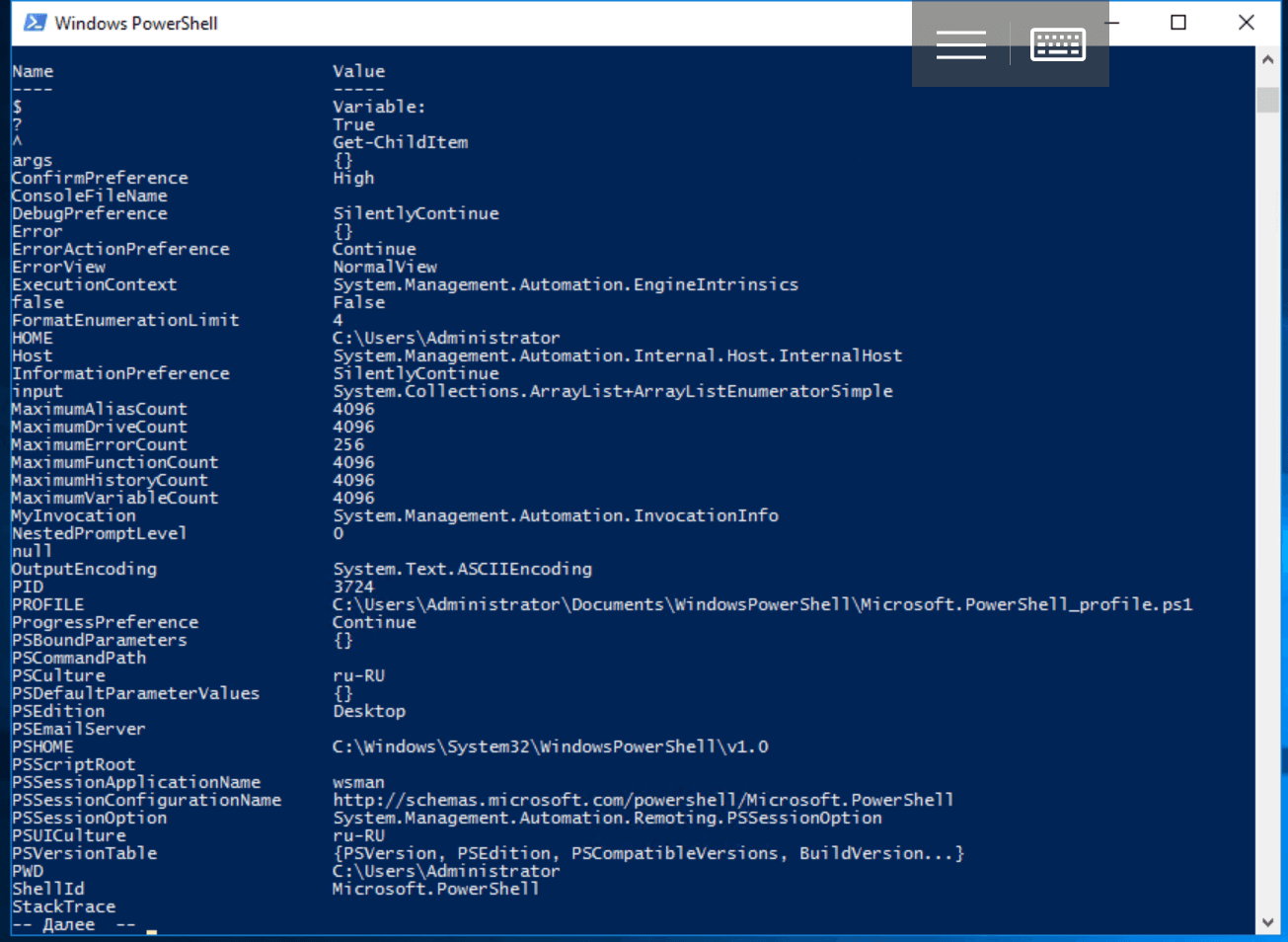
$PWD.Path

  
  
Для хранения пользовательских настроек нужны переменные предпочтений, значения которых можно изменить. К примеру, с помощью $ErrorActionPreference задается реакция интерпретатора команд на возникновение некритических ошибок.  
  
Вдобавок к операторам и командлетам для обращения к объявленным переменным существует псевдонакопитель Variable:. Работать с ним можно по аналогии с другими накопителями, а переменные в этом случае напоминают объекты файловой системы:

Get-ChildItem Variable: | more

или

ls Variable: | more



Области видимости

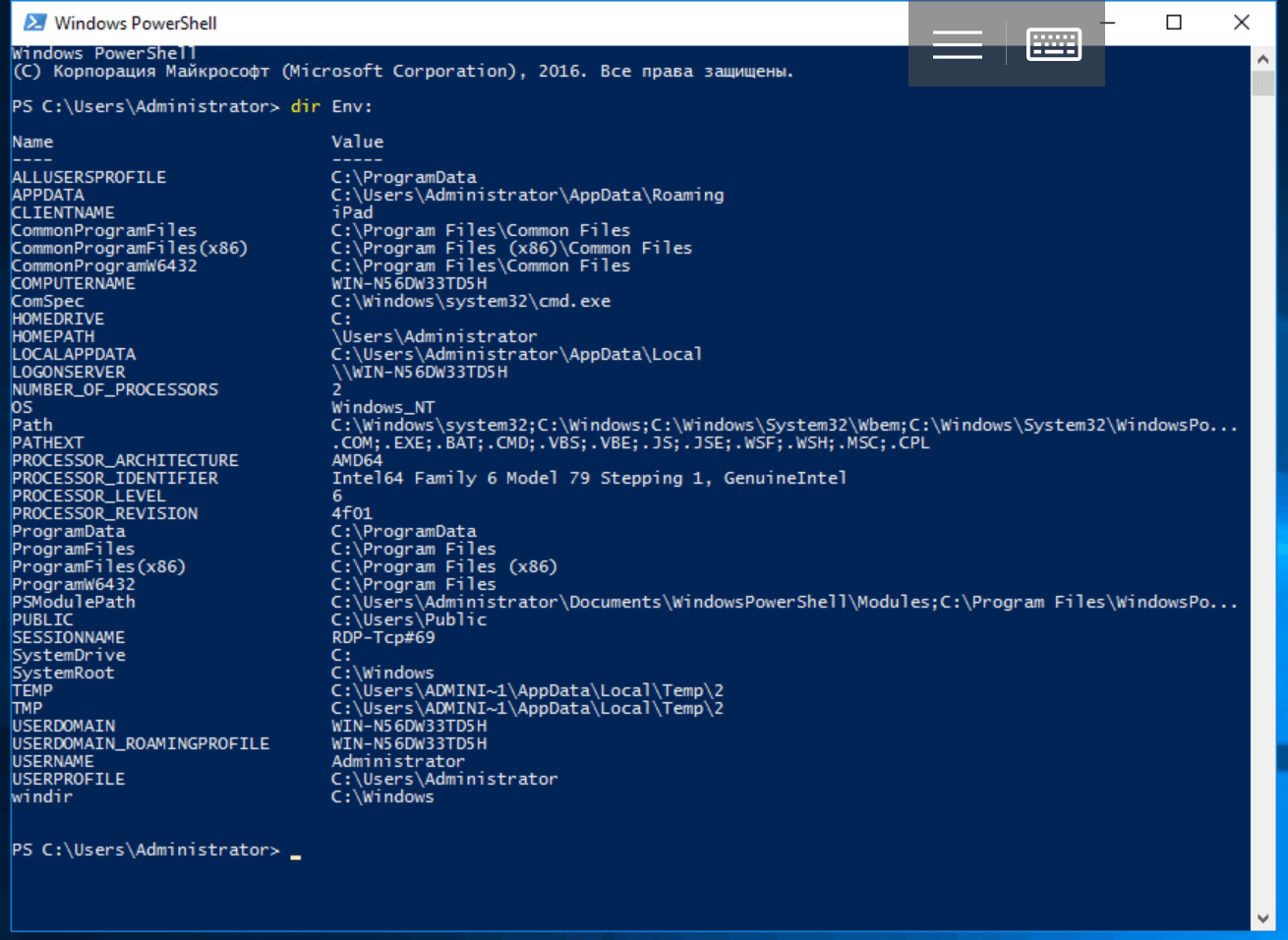
Для переменных в PowerShell существует понятие области видимости (Scope). Действие глобальной области (Global) распространяется на весь текущий сеанс — в нее входят, например, системные переменные. Локальные (Local) переменные доступны только в области, где они были определены: скажем внутри функции. Есть еще понятие области действия сценария (Script), но для команд скрипта она по сути является локальной. По умолчанию при объявлении переменных им задается локальная область действия, а чтобы это изменить, нужна специальная конструкция вида: $Global: переменная = значение.  
  
Например, так:

$Global:test = 100

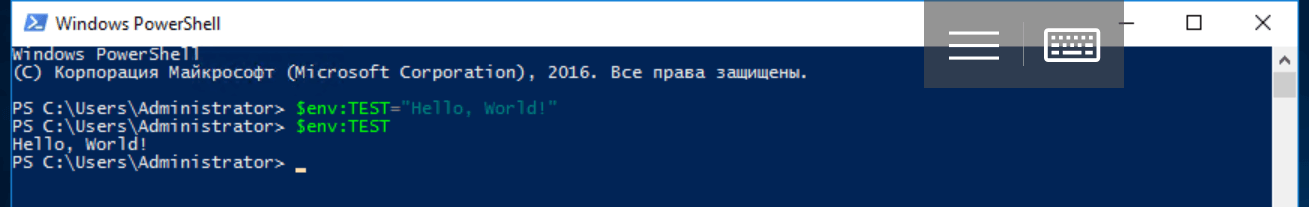
Переменные окружения (среды)

Из PowerShell доступен еще один псевдонакопитель Env:, с помощью которого можно обратиться к переменным среды. При запуске оболочки они копируются из родительского процесса (т.е. из инициировавшей текущий сеанс программы) и обычно их первоначальные значения совпадают со значениями в панели управления. Для просмотра переменных окружения используется командлет Get-ChildItem или его псевдонимы (алиасы): ls и dir.

dir Env:

  
  
Эти переменные представляют собой последовательности байтов (или символов, если угодно), интерпретация которых зависит только от использующей их программы. Командлеты \*-Variable с переменными среды не работают. Чтобы обратиться к ним, придется использовать префикс диска:

$env:TEST = "Hello, World!"



Арифметические операторы и операторы сравнения

В PowerShell есть следующие арифметические операторы: + (сложение), — (вычитание), \* (умножение), / (деление) и % (модуль или остаток от деления). Результат арифметического выражения вычисляется слева направо в соответствии с общепринятым порядком операций, а для группировки частей выражения применяются круглые скобки. Пробелы между операторами игнорируются, их используют только для облегчения восприятия. Оператор + также объединяет, а оператор \* повторяет строки. При попытке прибавить число к строке оно будет преобразовано в строку. Кроме того, в языке PowerShell есть множество операторов сравнения, которые проверяют соответствие между двумя значениями и возвращают логические True или False:

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Описание** | **Пример кода** |
| -eq | Equal / Равно (аналог = или == в других языках) | $test = 100 $test -eq 123 |
| -ne | Not equal / Не равно (аналог <> или !=) | $test = 100 $test -ne 123 |
| -gt | Greater than / Больше (аналог >) | $test = 100 $test -gt 123 |
| -ge | Greater than or equal / Больше или равно (аналог >=) | $test = 100 $test -ge 123 |
| -lt | Less than / Меньше (аналог <) | $test = 100 $test -lt 123 |
| -le | Less than or equal / Меньше или равно (аналог <=) | $test = 100 $test -le 123 |

Существуют и другие подобные операторы, позволяющие, например, сравнивать строки с учетом символа подстановки или использовать регулярные выражения для поиска соответствия образцу. Их мы подробно рассмотрим в следующих статьях. Символы <, > и = для сравнения не используются, поскольку задействованы для других целей.

Операторы присваивания

Помимо самого распространенного оператора = существуют и другие операторы присваивания: +=, -=, \*=, /= и %=. Они изменяют значение перед присвоением. Аналогично ведут себя унарные операторы ++ и --, которые увеличивают или уменьшают значение переменной — они тоже относятся к операторам присваивания.

Логические операторы

Для описания сложных условий одного только сравнения недостаточно. Записать любые логические выражения можно с помощью операторов: -and, -or, -xor, -not и!.. Работают они как и в других языках программирования, при этом можно использовать круглые скобки, чтобы задать порядок вычисления:

("Тест" -eq "Тест") -and (100 -eq 100)

-not (123 -gt 321)

!(123 -gt 321)

Условный переход

Операторы ветвления в PowerShell стандартные: IF(IF…ELSE, IF…ELSEIF…ELSE) и SWITCH. Рассмотрим их использование на примерах:

[int]$test = 100

if ($test -eq 100) {

Write-Host "test = 100"

}

[int]$test = 50

if ($test -eq 100) {

Write-Host "test = 100"

}

else {

Write-Host "test <> 100"

}

[int]$test = 10

if ($test -eq 100) {

Write-Host "test = 100"

}

elseif ($test -gt 100) {

Write-Host "test > 100"

}

else {

Write-Host "test < 100"

}

[int]$test = 5

switch ($test) {

0 {Write-Host "test = 0"}

1 {Write-Host "test = 1"}

2 {Write-Host "test = 2"}

3 {Write-Host "test = 3"}

4 {Write-Host "test = 4"}

5 {Write-Host "test = 5"}

default {Write-Host "test > 5 или значение не определено"}

}

Циклы

В языке PowerShell есть несколько разновидностей циклов: WHILE, DO WHILE, DO UNTIL, FOR и FOREACH.  
  
Цикл с предусловием работает, если/пока оно выполняется:

[int]$test = 0

while ($test -lt 10) {

Write-Host $test

$test = $test + 1

}

Циклы с постусловием отработают хотя бы один раз, потому что проверка условия производится после выполнения итерации. При этом DO WHILE работает, пока условие истинно, а DO UNTIL — пока оно ложно:

[int]$test = 0

do {

Write-Host $test

$test = $test + 1

}

while ($test -lt 10)

[int]$test = 0

do {

Write-Host $test

$test = $test + 1

}

until ($test -gt 9)

Количество итераций цикла FOR известно заранее:

for ([int]$test = 0; $test -lt 10; $test++) {

Write-Host $test

}

В цикле FOREACH осуществляет перебор элементов массива или коллекции (хэш-таблицы):

$test\_collection = "item1", "item2", "item3"

foreach ($item in $test\_collection)

{

Write-Host $item

}

Массивы

В переменных PowerShell хранятся не только единичные объекты (число, строка и т.д.), но и множественные. Самая простая разновидность таких переменных — массивы. Массив может состоять из нескольких элементов, из одного элемента или быть пустым, т.е. не содержать элементов. Для его объявления используется оператор @(), который понадобится нам в следующей статье — он очень важен для добавления в массив других массивов (создания многомерных массивов), передачи массивов в функции в качестве аргумента и тому подобных задач:

$test\_array = @() *#создаем пустой массив*

При инициализации массива его значения перечисляются через запятую (специальный оператор ,):

$test\_array = @(1, 2, 3, 4) *# создаем массив из четырех элементов*

В большинстве случаев оператор @() можно опустить:

$test\_array = 1, 2, 3, 4

В этом случае массив из одного элемента инициализируется следующим образом

$test\_array = , 1

Для обращения к элементам массива используется начинающийся с нуля целочисленный индекс и оператор индекса (квадратные скобки):

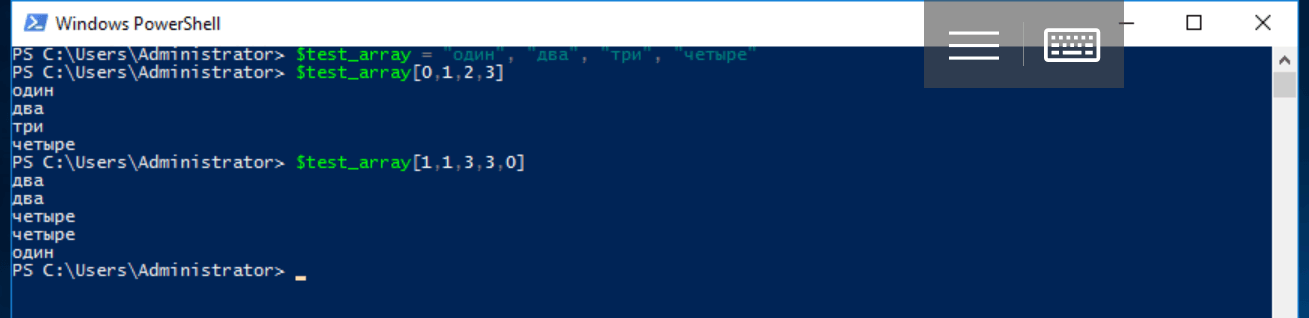
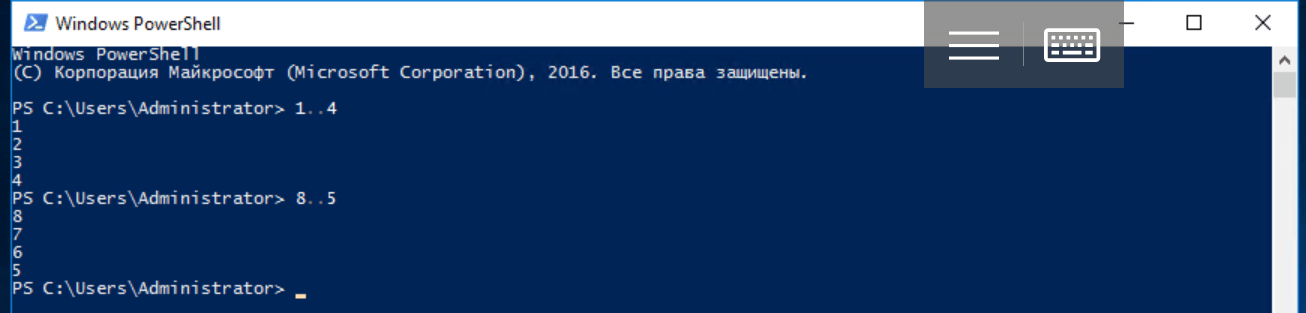
$test\_array[0] = 1

Можно указать несколько индексов через запятую, в т.ч. повторяющихся:

$test\_array = "один", "два", "три", "четыре"

$test\_array[0,1,2,3]

$test\_array[1,1,3,3,0]

  
  
Оператор .. (две точки — оператор диапазона) возвращает массив целых чисел на определенном верхней и нижней границей отрезке. Например, выражение 1..4 выводит массив из четырех элементов @(1, 2, 3, 4), а выражение 8..5 — массив @(8, 7, 6, 5).  
  
  
  
С помощью оператора диапазона можно инициализировать массив ($test\_array = 1..4) или получить срез (slice), т.е. последовательность элементов одного массива с индексами из другого. При этом отрицательное число -1 обозначает последний элемент массива, -2 — предпоследний и т.д.

$test\_array = "один", "два", "три", "четыре"

$test\_array[0..2]

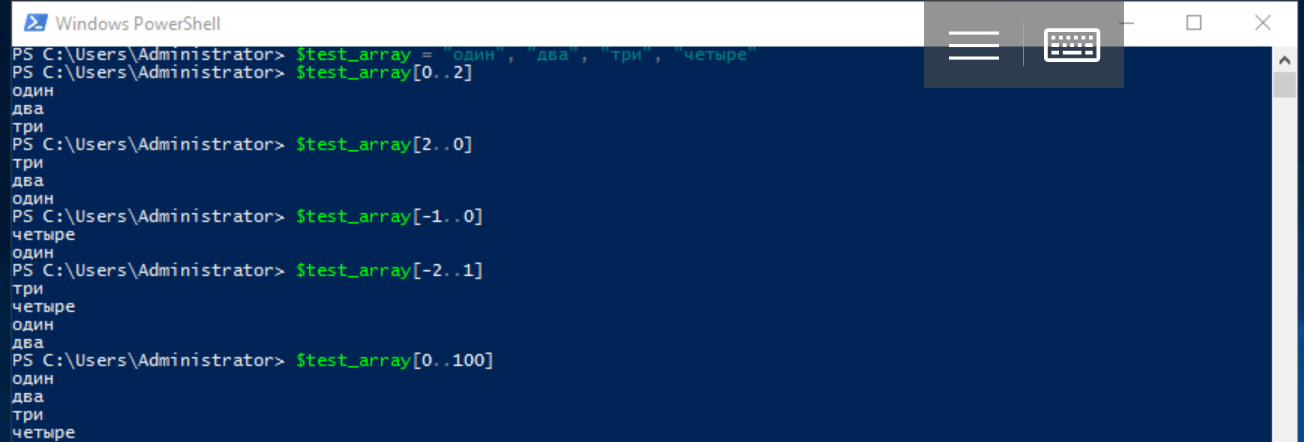
$test\_array[2..0]

$test\_array[-1..0]

$test\_array[-2..1]

Обратите внимание, что значения целочисленного массива могут быть больше максимального значения индекса массива с данными. В этом случае возвращаются все значения до последнего:

$test\_array[0..100]

Если попытаться обратиться к единственному несуществующему элементу массива, возвращается значение $null.  
  
  
  
В PowerShell массивы могут содержать элементы разных типов или быть строго типизированными:

$test\_array = 1, 2, "тест", 3, 4

for ([int]$i = 0; $i -lt $test\_array.count; $i++)

{

Write-Host $test\_array[$i]

}

Где свойство $test\_array.count — количество элементов массива.  
  
Пример создания строго типизированного массива:

[int[]]$test\_array = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Хэш-таблицы

Еще один базовый тип переменных в языке PowerShell — хэш-таблицы, которые также называют ассоциативными массивами. Hashtable похожи на JSON object и строятся по принципу ключ-значение. В отличие от обычных массивов, доступ к их элементам осуществляется по именованным ключам, которые являются свойствами объекта (также можно использовать оператор индекса — квадратные скобки).  
  
Пустая хэш-таблица объявляется с помощью служебного символа @ и операторных скобок:

$test\_hashtable = @{}

При объявлении можно сразу создать ключи и присвоить им значения:

$test\_hashtable = @{one="один"; two="два"; three="три"; "some key"="some value"}

Для добавления элемента в хэш-таблицу нужно присвоить ей еще несуществующий ключ или воспользоваться методом Add(). Если присваивание делается с существующим ключом, его значение изменится. Для удаления элемента из хэш-таблицы используется метод Remove().

$test\_hashtable."some key"

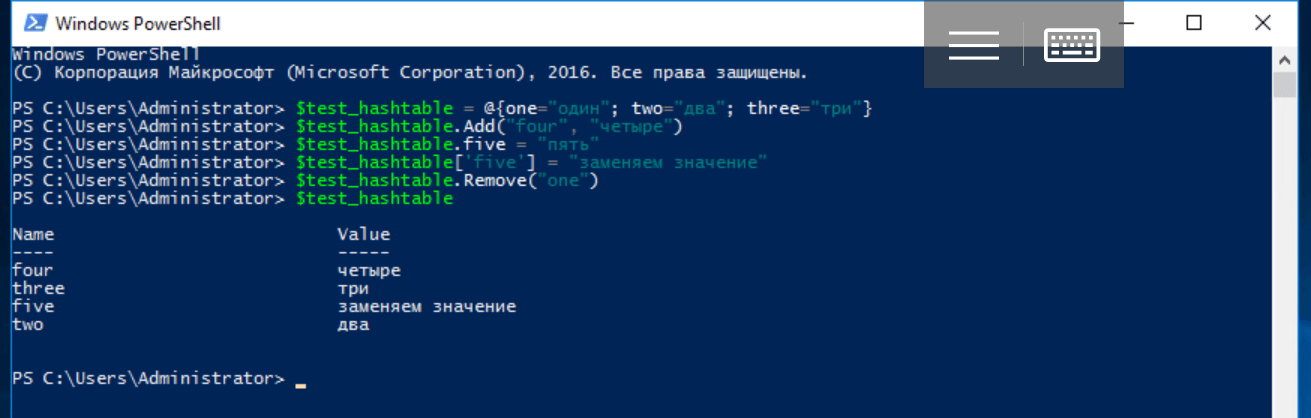
$test\_hashtable["some key"]

$test\_hashtable.Add("four", "четыре")

$test\_hashtable.five = "пять"

$test\_hashtable['five'] = "заменяем значение"

$test\_hashtable.Remove("one")

  
  
Переменные этого типа можно передавать в качестве аргументов функциям и командлетам — в следующей статье мы изучим как это делается, а также рассмотрим еще один сходный тип — PSCustomObject.

Функции

В языке PowerShell есть все необходимые для процедурного программирования элементы, включая функции. Для их описания используется служебное слово Function, после которого требуется указать имя функции и заключенное в операторные скобки тело. При необходимости передать в функцию аргументы их можно указать сразу после имени в круглых скобках.

function имя-функции (аргумент1, ..., аргументN)

{

тело-функции

}

Функция всегда возвращает результат — это массив результатов всех ее стейтментов, если их более одного. Если стейтмент один, возвращается единственное значение соответствующего типа. Конструкция return $value добавляет элемент со значением $value к массиву результатов и прерывает выполнение statement list, а пустая функция возвращает $null.  
  
Для примера создадим функцию возведения числа в квадрат:

function sqr ($number)

{

return $number \* $number

}

Отметим, что в теле функции можно использовать любые объявленные до ее вызова переменные, а вызов функций в PowerShell может показаться непривычным: аргументы (если они есть) не заключаются в круглые скобки и разделяются пробелами.

sqr 2

или так:

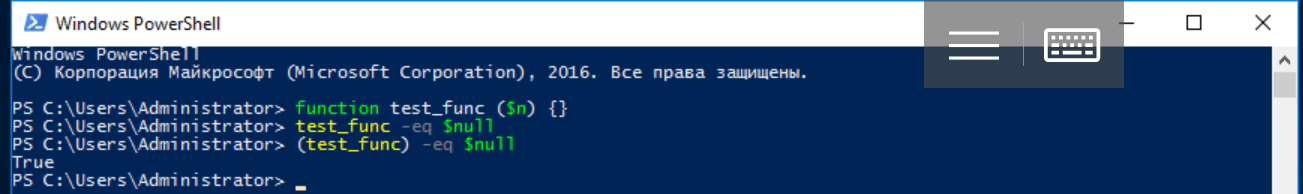
sqr -number 2

Из-за способа передачи аргументов саму функцию иногда приходится заключать в скобки:

function test\_func ($n) {}

test\_func -eq $null *# функция не вызывалась*

(test\_func) -eq $null *# результат выражения — $true*

  
  
При описании функции можно присвоить аргументам значения по умолчанию:

function func ($arg = value) {

*#тело функции*

}

Существует и другой синтаксис для описания аргументов функции, кроме того параметры могут считываться из конвейера — все это пригодится в следующей статье, когда мы будем рассматривать экспортируемые модули и создание собственных командлетов.

Обработка ошибок

В PowerShell существует механизм Try…Catch…Finally, позволяющий обрабатывать исключительные ситуации. В блок Try помещается код, в котором может возникнуть ошибка, а в блок Catch — ее обработчик. Если ошибки не было, он не выполняется. Блок Finally выполняется после блока Try вне зависимости от возникновения ошибки, а блоков Catch может быть несколько для исключений различных типов. Само исключение записывается в не требующую объявления переменную по умолчанию ($\_) и может быть легко извлечено. В примере ниже мы реализуем защиту от ввода некорректного значения:

try {

[int]$test = Read-Host "Введите число"

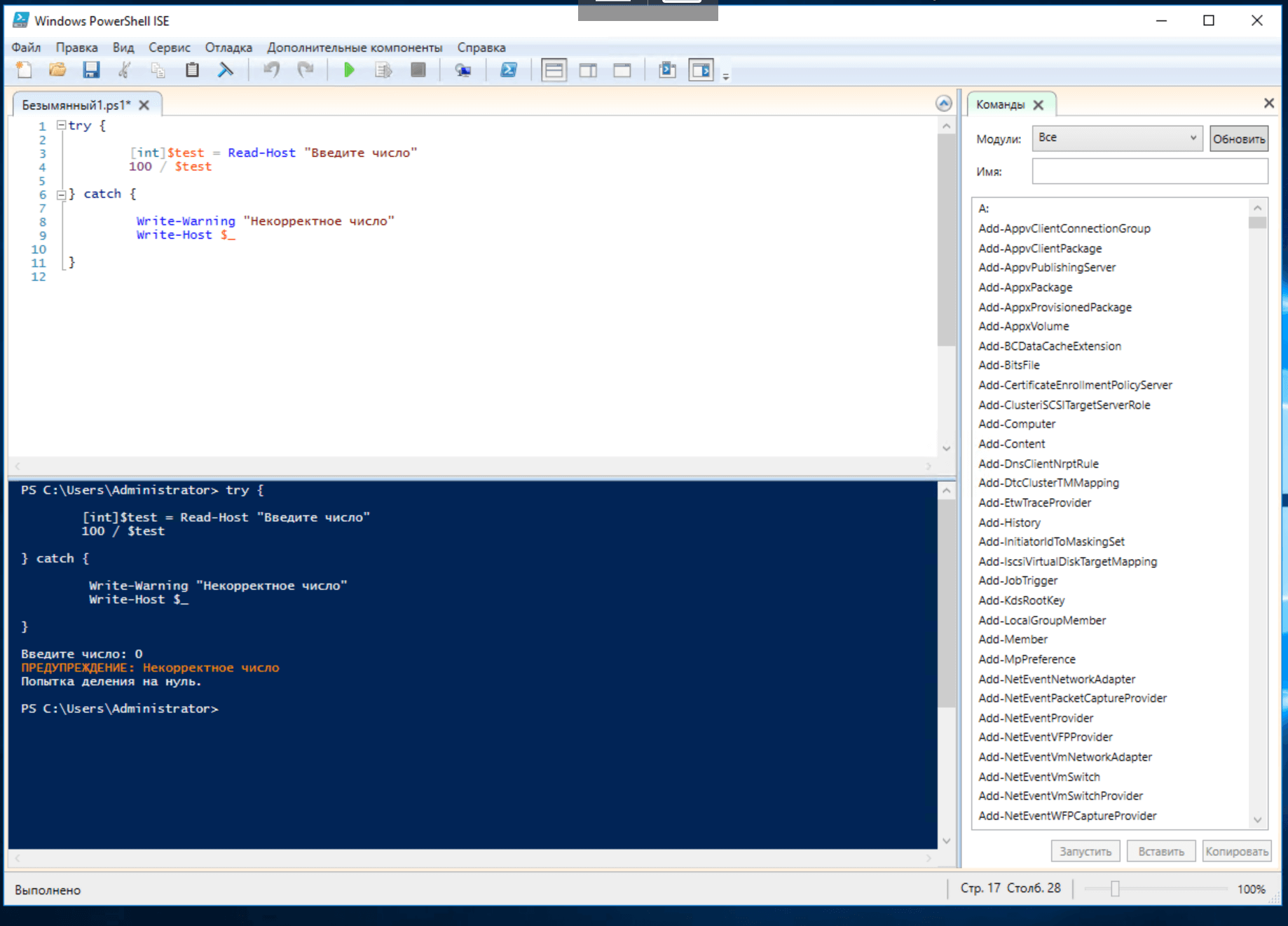
100 / $test

} catch {

Write-Warning "Некорректное число"

Write-Host $\_

}

  
  
На этом рассмотрение основ программирования на языке PowerShell стоит закончить. В следующих статьях мы подробнее изучим работу с переменными разных типов, коллекции, регулярные выражения, создание функций, модулей и собственных командлетов, а также объектно-ориентированное программирование.