Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

Г. В. Горбань

Засоби автоматизації завдань в операційній системі Windows

Навчальний посібник

Миколаїв – 2021

УДК 004.451(075.8)
Г 67

Рекомендовано до друку вченою радою Чорноморського національного університету імені Петра Могили (протокол № 5 від 10 червня 2021 р.).

Рецензенти:

Любченко В. В., доктор технічних наук, професор кафедри системного ПЗ Державного університету «Одеська політехніка»;

Райко Г. О,. кандидат технічних наук, доцент, завідувач кафедри інформаційних технологій Херсонсь-кого національного технічного університету.

Г 67 **Горбань Г. В.** Засоби автоматизації завдань в операційній системі Windows : навч. посіб. / Г. В. Горбань. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2021. – 168 с.

ISBN 978-966-336-427-8

У посібнику розглянуто основи роботи з інстументами автоматизації рутинних задач у операційній системі Windows. Посібник складається з трьох розділів, кожен з яких присвячений окремому інструменту. У першому розділі розглянуто основні команди оболонки cmd та основи програмування пакетних командних файлів (.bat). Другий розділ присвячений серверу сценаріїв WSH та основам програмування скриптів мовою Jscript для їх виконання в операційній системі. У третьому розділі розглядається відносно новий інструмент, яким є командна оболонка Windows PowerShell. Також розглянуто основи мови програмування сценаріїв цієї командної оболонки. Посібник призначено для студентів спеціальностей 121 – «Інженерія програмного забезпечення», а також він може бути корисним для студентів інших спеціальностей галузі знань 12 – «Інформаційні технології».

УДК 004.451(075.8)

ISBN 978-966-336-427-8

© Горбань Г. В., 2021
© ЧНУ ім. Петра Могили, 2021

Зміст

ВСТУП **6**
РОЗДІЛ 1. ОБОЛОНКА КОМАНДНОГО РЯДКА WINDOWS. ІНТЕРПРЕТАТОР CMD.EXE 13
1.1. Внутрішні і зовнішні команди. Структура команд 13
1.1.1. Перенаправлення введення / виведення і конвеєризація (композиція) команд 15
1.1.2. Команди MORE і SORT 17
1.1.3. Умовне виконання і угруповання команд............................ 18
1.2. Команди для роботи з файловою системою 19
1.3. Мова інтерпретатора Cmd.exe. Командні файли 27
1.3.1. Виведення повідомлень і дублювання команд.................... 27
1.3.2. Використання параметрів командного рядка 30
1.3.3. Робота зі змінними 33
1.3.4. Призупинення виконання та виклик командних файлів ... 36
1.4. Оператори переходу у командних файлах 37
1.4.1. Оператори умови у командних файлах 38
1.4.2. Організація циклів у командних файлах............................. 42
РОЗДІЛ 2. СЕРВЕР СЦЕНАРІЇВ WINDOWS SCRIPT HOST........50
2.1. Створення і запуск найпростіших сценаріїв WSH 52
2.2. Мови VBScript і JScript для сценаріїв WSH.................................. 54
2.2.1. Основні відомості про мову Jscript...................................... 56
2.3. Власна об’єктна модель WSH 60
2.3.1. Об’єкт WScript... 62
2.3.2. Об’єкт WshShell 67
2.3.3. Об’єкт WshShortcut 73
2.3.4. Об’єкти-колекції 74
2.3.5. Об’єкт WshEnvironment.. 76
2.3.6. Об’єкт WshSpecialFolders 77
2.4. Сценарії WSH для доступу до файлової системи. Об’єктна модель FileSystemObject..77
2.4.1. Об’єкт FileSystemObject 79
2.4.2. Об’єкт Drive... 81
2.4.3. Об’єкт Folder 83
2.4.4. Об’єкт File.. 85
2.4.5. Приклади сценаріїв 86
РОЗДІЛ 3. КОМАНДНА ОБОЛОНКА WINDOWS POWERSHELL **91**
3.1. Типи команд PowerShell 94

3.1.1. Імена та структура командлетів 95
3.1.2. Псевдоніми команд 96
3.1.3. Довідкова система PowerShell 97
3.2. Робота з файловою системою у PowerShell 98
3.3. Конвеєризація об’єктів у PowerShell 102
3.3.1. Командлет Get-Member – перегляд структури об’єктів 105
3.3.2. Командлет Where-Object – фільтрація об’єктів 106
3.3.3. Командлет Sort-Object – сортування об’єктів 108
3.3.4. Командлет Select-Object – виділення об’єктів та властивостей 110
3.3.5. Командлет ForEach-Object – виконання довільних дій над об’єктами у конвеєрі 112
3.3.6. Командлет Group-Object – групування об’єктів 113
3.3.7. Командлет Measure-Object – вимірювання характеристик об’єктів 113
3.4. Управління виведенням команд у PowerShell 114
3.4.1. Форматування виведеної інформації 115
3.4.2. Перенаправлення інформації, що виводиться 117
3.4.3. Збереження даних у файл 118
3.4.4. Друк даних та придушення виведення 119
3.5. Програмування сценаріїв командного рядка у Windows PowerShel 120
3.5.1. Символьні рядки 121
3.5.2. Змінні PowerShell 124
3.5.3. Масиви у PowerShell 127
3.5.4. Хеш-таблиці (асоціативні масиви) 131
3.6. Оператори у PowerShell 134
3.6.1. Арифметичні оператори 134
3.6.2. Оператори перевірки на відповідність шаблону 138
3.6.3. Логічні оператори 140
3.7. Керуючі інструкції мови PowerShell 141
3.7.1. Інструкція If ... ElseIf ... Else 141
3.7.2. Цикл While 142
3.7.3. Цикл Do ... While 143
3.7.4. Цикл ForEach 143
3.7.5. Мітки циклів, інструкції Break і Continue 145
3.7.6. Інструкція Switch 146
3.8. Функції у PowerShell 150
3.8.1. Обробка аргументів функцій за допомогою змінної $Args 151
3.8.2. Формальні параметри функцій 152

3.8.3. Значення, що повертаються 156
3.8.4. Функції всередині конвеєра команд 157
3.8.5. Фільтри в PowerShell 157
3.9. Сценарії PowerShell 158
3.9.1. Передача аргументів у сценарії 160
3.10. Доступ з PowerShell до зовнішніх об'єктів 161
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 165

Вступ

У наш час графічний інтерфейс Windows став настільки звичним, що багато користувачів та адміністраторів з невеликим досвідом навіть не замислюються (а часто навіть не знають) про різні рішення для управління операційною системою, що є альтернативами командному рядку та можуть автоматично вирішувати різні проблеми без ручного втручання. Ця ситуація зумовлена таким фактом: історично командний рядок був слабким місцем операційної системи Windows (порівняно з системами UNIX). Основною причиною цього є те, що спочатку Microsoft зосередилася на багатьох недосвідчених користувачах, які не бажали вникати в технічні деталі виконання певних операцій у системі. Тому основні зусилля розробників операційних систем спрямовані на вдосконалення графічної оболонки, щоб зробити роботу непрофесіоналів більш комфортною, а не на створення робочого середовища для професіоналів або досвідчених користувачів.

Як показав час, ця стратегія виявилася успішною з комерційної точки зору на ринку персональних (домашніх чи офісних) комп'ютерів – мільйони людей використовують графічний інтерфейс Windows для запуску необхідних програм. Більше того, керувати сервером Windows сьогодні дуже просто – операційна система надає зручні графічні інструменти для налаштування та виконання щоденних завдань управління. За допомогою служб терміналів ви можете легко фізично працювати на віддаленому сервері, який знаходиться на іншому континенті.

Однак ця модель управління не є розширюваною: якщо ви використовуєте стандартні графічні інструменти для управління одним сервером замість десяти серверів, одну і ту ж послідовність змін управління у діалоговому вікні потрібно повторити десять разів, тому в цьому випадку виникнуть проблеми автоматизації. Звичайні операції на багатьох комп'ютерах (наприклад, інвентаризація обладнання та програмного забезпечення, моніторинг служб, аналіз журналів подій тощо) можуть бути виконані за допомогою спеціальних (зазвичай важких і недешевих) застосунків (наприклад, Microsoft Systems Management Server (SMS)) або сценаріїв, які написані адміністратором (мовою командного рядка або спеціальною мовою сценаріїв) і безпосередньо підтримуються операційною системою. У другому випадку не потрібно встановлювати сторонні програмні продукти.

Тому професіоналам, які керують інформаційними системами на базі Windows, потрібно лише розуміти функції командного рядка,

сценаріїв та технологій автоматизації, що підтримуються цією операційною системою.

Однак помилково думати, що лише адміністраторам потрібні командні рядки або сценарії. Зрештою, щоденні завдання користувача, які зазвичай можна виконувати за допомогою графічного інтерфейсу провідника Windows (наприклад, копіювання чи архівування файлів, підключення або відключення мережевих ресурсів тощо), можна повністю автоматизувати, написавши простий пакетний код командного файлу, що буде містити лише кілька рядків. Більше того, для тих, хто не розуміє основних команд Windows та основних функцій операційної системи, таких як перенаправлення введення / виведення, і конвеєєрів, деякі найпростіші завдання не здаються тривіальними. Наприклад, спробуйте використовувати лише графічні інструменти для створення файлу, що містить імена файлів усіх підкаталогів з будь-якого каталогу. Однак для цього достатньо виконати уніфіковану команду DIR (з використанням відповідних ключів) і перенаправити результат цієї команди у необхідний текстовий файл.

Якими ми хочемо бачити інструменти, що використовуються для автоматизації завдань в операційній системі, і які функції вона повинна мати? Бажано, щоб вони мали такі функції:

– запуск у різних версіях операційної системи (бажано у всіх) без встановлення будь-якого іншого програмного забезпечення;
– інтеграція з командною оболонкою;
– узгоджений та сумісний синтаксис команд та утиліт;
– існування детальної вбудованої довідки про команди та приклади використання;
– можливість виконувати сценарії, написані простою для вивчення мовою;
– можливість використання всіх технологій, що підтримуються операційною системою.

У системах UNIX стандартні програми оболонки (bashell, kshell, cshell тощо) виступають інструментами автоматизації, і цей аспект операційної системи стандартизований у стандарті POSIX (Mobile System Standard).

В операційній системі Windows ситуація складніша. Поки що не існує «ідеального» інструменту автоматизації Windows, який може відповідати всім вищезазначеним вимогам. Остання версія операційної системи підтримує кілька стандартних засобів автоматизації, які сильно відрізняються один від одного: оболонка командного рядка ***cmd.exe***, середовище сценаріїв ***Windows Script Host (WSH)*** і оболонка

Windows PowerShell. Тому адміністратори або користувачі Windows повинні вибрати, який метод використовувати для вирішення конкретних проблем, тому вони повинні чітко розуміти переваги та недоліки цих автоматизованих інструментів. Далі ми коротко обговоримо переваги та недоліки кожного методу.

Усі версії операційної системи Windows підтримують інтерактивні оболонки командного рядка, а за замовчуванням встановлено набір утиліт командного рядка (їх кількість та склад залежить від версії операційної системи). Як правило, будь-яка операційна система може бути представлена у сукупності ядра системи, що може отримувати доступ до апаратного забезпечення та керувати файлами і процесами, також оболонкою з утилітами, що дозволяють користувачам отримувати доступ до функцій ядра операційної системи. Механізм роботи оболонки в різних системах однаковий: у відповідь на запрошення (підказку), що виводиться оболонкою, користувач вводить команду (функцію цієї команди може реалізувати сама оболонка або зовнішня утиліта), потім оболонка виконує цю операцію і відображає певну інформацію за необхідності. Командна оболонка знову виводить запрошення і чекає введення користувачем наступної команди.

З технічної точки зору програма-оболонка є простим порядковим інтерпретатором для директивного програмування і може використовуватися як оператор виконуваних програм.

Зазвичай, разом з інтерактивним режимом роботи, командна оболонка підтримує і пакетний режим, у якому система послідовно виконує команди, написані у сценаріях текстових файлів. Оболонка Windows не є винятком. Що стосується програмування, то мову командних файлів Windows можна охарактеризувати таким чином:

- впровадження директивної парадигми програмування;
- виконання в режимі порядкової інтерпретації;
- наявність керуючих конструкцій;
- підтримка кількох типів циклів (включаючи спеціальні цикли для обробки текстових файлів);
- існування оператора присвоєння значення змінної;
- можливість використання зовнішніх програм (команд) операційної системи як операторів та обробляти їх коди повернення;
- наявність нетипізованих змінних, які декларуються першою згадкою (значення змінних можна інтерпретувати як числа та використовувати в цілочисельних арифметичних виразах).

Починаючи з версії Windows NT, командна оболонка представлена інтерпретатором ***cmd.exe***, який розширює функціональні можливості оболонки ***command.com***, що використовувалась в операційній системі

MS-DOS. У свою чергу, функція інтерпретатора команд ***command.com*** запозичена у операційної системи CP/M, оболонка якої є спрощеною версією оболонки систем сімейства UNIX.

Тому з точки зору простоти використання та розвитку мов сценаріїв оболонка командного рядка MS-DOS спочатку поступалася оболонкам UNIX-подібних систем. У оболонці Windows (***cmd.exe***), незважаючи на всі вдосконалення, все ще неможливо подолати це відставання як інтерактивному режимі (наприклад, cmd.exe не підтримує псевдоніми для довгих імен команд і не має реалізації автоматичного завершення імені команди у разі введення їх назв), так і в синтаксисі мови командних файлів. Windows завжди програвала системам UNIX за кількістю та характеристиками стандартних інструментів командного рядка, а також за якістю та повнотою вбудованої довідкової системи для команд оболонки.

На практиці проблему недостатньої функціональності стандартних команд потрібно вирішувати за допомогою утиліти Windows Resource Kit відповідної версії операційної системи, або для її вирішення потрібно знайти відповідну сторонню утиліту. Крім того, можна використовувати пакет Microsoft Services For Unix (SFU) у Windows, щоб використовувати утиліти та оболонки, сумісні з POSIX. Цей продукт був розроблений для Windows NT і спочатку не входив до складу операційної системи, і його потрібно було придбати за окрему плату. Пізніше пакет програм SFU став безкоштовним і навіть включений в операційну систему Windows Server 2003 R2.

Отже, з огляду на всі вищезазначені обставини, можна зробити висновок, що командна оболонка ***cmd.exe*** та командні файли – це найбільш прості в освоєнні засоби автоматизації в Windows. Вони можуть використовуватися у всіх версіях операційних систем, але вони значною мірою програють відповідним інструментам у UNIX-подібних системах. Також не надається доступ до об’єктних моделей, що підтримуються операційною системою (COM, WMI, .NET). Це теж є істотним недоліком командної оболонки ***cmd.exe***.

Наступним кроком у розвитку інструментів та технологій автоматизації завдань в операційній системі Windows є поява сервера сценаріїв Windows Script Host (WSH). Цей інструмент доступний для всіх версій Windows і дозволяє запускати сценарії, написані повноцінними мовами сценаріїв (за замовчуванням, VBScript та JScript) безпосередньо в операційній системі. Раніше ці мови програмування могли були використані лише на HTML-сторінках і могли працювати в контексті безпеки веббраузера. У таких випадках не можна було отримати доступ до файлової системи локального комп’ютера.

Порівняно з пакетними командними файлами сценарії WSH мають ряд переваг. Перш за все, VBScript та JScript – це повноцінні алгоритмічні мови із вбудованими функціями та методами для обробки рядків, виконання математичних операцій, обробки виключень тощо; крім того, може бути використана будь-яка інша мова сценаріїв із встановленими відповідними модулями підтримки (наприклад, Perl, Python, Ruby) для написання сценаріїв WSH.

По-друге, WSH підтримує декілька власних об'єктів. Властивості та методи цих об'єктів дозволяють вирішувати деякі типові щоденні завдання адміністраторів операційних систем: використовувати мережеві ресурси, змінні середовища, реєстр, ярлики та спеціальні папки Windows, запускати та керувати іншими програмами.

По-третє, зі сценарію WSH можете отримати доступ до служб будь-якого автоматизованого серверного додатка (наприклад, програми Microsoft Office), об'єкти якого зареєстровані в операційній системі.

Нарешті, сценарій WSH дозволяє використовувати об'єкти інформаційної моделі Windows Management Instrumentation (WMI), який надає програмний інтерфейс для управління всіма компонентами операційної системи, а також служби каталогів Active Directory Service Interface (ADSI).

Слід також зазначити, що Windows давно підтримує технологію WSH. В Інтернеті (включаючи вебсайт Microsoft) ви можете знайти безліч готових сценаріїв, які виконують певні операції та мають певні навички та знання, щоб швидко зробити ці сценарії придатними для виконання своїх конкретних завдань.

Поговоримо тепер про слабкі місця WSH. Перш за все, сам по собі WSH – це тільки середовище виконання сценаріїв, а не оболонка. По-друге, WSH не інтегрований з командним рядком, тобто відсутній режим, в якому можна вводити команди з клавіатури і відразу бачити результат їх виконання.

Великим недоліком є те, що операційна система за замовчуванням не має повної та детальної довідкової інформації про об'єкти WSH та мови VBScript та JScript (потрібно шукати документацію в Інтернеті на вебсайті Microsoft). Іншими словами, наприклад, якщо ви не пам'ятаєте синтаксис певного методу VBScript або JScript або точну назву властивості об'єкта WSH, під рукою у вас немає роздрукованого документа, і ваш комп'ютер не може отримати доступ до Інтернету, то написати правильний сценарій буде просто неможливо (у зв'язку з цим командні файли є більш універсальними, оскільки майже всі команди мають принаймні вбудований опис використовуваних ними ключів, а операційна система має довідкову систему, що містить інформацію про всі стандартні команди).

Нарешті, сценарії WSH створюють серйозну потенційну загрозу безпеці, оскільки відомо, що багато вірусів використовують WSH для виконання руйнівних операцій.

Тому ми можемо дати таку загальну оцінку: сценарії WSH – це універсальний інструмент, що у будь-якій версії операційної системи Windows може вирішити проблеми автоматизації майже будь-якого ступеня складності, але для вивчення сценаріїв потрібно багато роботи з вивчення мови програмування та багатьох суміжних технологій управління операційною системою (WMI, ADSI тощо).

Тому до початку XXI століття стан засобів автоматизації Windows не можна назвати дуже хорошим. З одного боку, функціональність і гнучкість мови оболонки cmd.exe очевидно недостатні, з іншого боку, виявляється, що сценарії WSH, які використовуються з об'єктними моделями ADSI та WMI, занадто складні для звичайних користувачів та адміністраторів-початківців.

У 2000 році розпочалася розробка нової оболонки для доступу до об'єктів WMI з командного рядка (командний рядок WMI, WMIC). Цей продукт не має особливого успіху, оскільки він орієнтований на функції WMI, а не на зручність користувачів. Коли почали вдосконалювати WMIC, у Microsoft зрозуміли, що можна реалізувати програму оболонки, яка б не обмежувалась використанням об'єктів WMI, а могла також отримати доступ до будь-якого класу .NET Framework, надаючи тим самим можливість використовувати у командному рядку усі потужні функції цього середовища.

Розробники нової оболонки під назвою Windows PowerShell (раніше відома як Monad) мали такі основні цілі, що були успішно вирішені:

- використання командного рядка як основного інтерфейсу управління;
- впровадження моделі ObjectFlow (елементом обміну інформацією є об'єкт);
- редизайн існуючих команд, утиліт та програм оболонки;
- інтеграція командного рядка з об'єктами COM, WMI та .NET;
- використання будь-якого джерела даних у командному рядку відповідно до принципу файлової системи.

Взагалі найважливішою ідеєю PowerShell є те, що результатом командного рядка результату команди є не текст (представлений у порядку байтів), а об'єкт (дані та властиві йому методи). Як результат, працювати в PowerShell простіше, ніж у традиційній оболонці, тому що для вилучення необхідної інформації з потоку символів не потрібно нічого робити.

Крім того, розробники намагалися зібрати всі найкращі аспекти інших оболонок командного рядка з різних операційних систем у PowerShell. Вони вважають, що такі продукти мали значний вплив на PowerShell:

- BASH, KSH (конвеєризація або композиція команд);
- AS400/VMS (стандартна назва команди для прискорення швидкості навчання);
- TCL/WSH (можливість вбудовування та декількох мов);
- PERL, PYTHON (виразність і стиль).

Звернемо увагу, що PowerShell – це оболонка командного рядка (користувачі працюють інтерактивно) та середовище сценаріїв, написане спеціальною мовою PowerShell. Інтерактивний сеанс у PowerShell подібний до роботи в оболонці системи UNIX: усі команди в PowerShell мають детальну вбудовану довідку (для більшості команд представлені приклади їх використання) та підтримку автоматичного заповнення імені команди та її параметрів, коли команда вводиться з клавіатури. Також у PowerShell є підтримка псевдонімів, подібних до назв службових програм UNIX (ls, pwd, tee тощо).

Порівняно з VBScript або JScript, мову PowerShell легше засвоювати і простіше писати сценарії, що мають доступ до зовнішніх об'єктів. Особлива увага приділяється проблемам безпеки під час використання сценаріїв (наприклад, сценарії можна запускати лише за допомогою повного шляху до сценарію, а за замовчуванням сценарії PowerShell зазвичай забороняється запускати в системі). Як правило, PowerShell є більш зручним і потужним за своїх попередників (cmd.exe та WSH), а головним недоліком, який заважає поширенню нових інструментів, є те, що PowerShell не може працювати у всіх версіях операційної системи Windows. Оболонку можна використовувати лише у версії не нижче Windows XP із пакетом оновлень 2 зі встановленим .NET Framework 2.0.

Розділ 1.
Оболонка командного рядка Windows.
Інтерпретатор cmd.exe

В операційній системі Windows, як і в інших операційних системах, для виконання інтерактивних команд використовується так званий інтерпретатор команд (також іменований як командний процесор або оболонка командного рядка). Командний інтерпретатор або оболонка командного рядка – це програма, яка може читати та обробляти набрані команди в пам’яті. У Windows 9x, як і в MS-DOS, оболонка за замовчуванням представлена виконуваним файлом command.com. Починаючи з версії Windows NT, операційна система реалізує інтерпретатор команд cmd.exe, який має більш потужні функції.

У Windows, як і інші виконувані файли, що відповідають командам зовнішньої операційної системи, файл cmd.exe знаходиться в каталозі %SystemRoot%\SYSTEM32 (значення змінної середовища %SystemRoot% – це системний каталог Windows, зазвичай C:\Windows або C:\WinNT). Щоб запустити оболонку (відкрити новий сеанс командного рядка), потрібно обрати «Виконати ...» в меню «Пуск», потім увести ім’я файлу Cmd.exe та натиснути «OK». У результаті відкриється нове вікно, де можна запускати команди та переглядати результати їх роботи.

1.1. Внутрішні і зовнішні команди. Структура команд

Деякі команди безпосередньо розпізнаються та виконуються самим інтерпретатором команд – ці команди називаються внутрішніми командами (наприклад, COPY або DIR). Інші команди операційної системи є окремими програмами і за замовчуванням знаходяться в тому ж каталозі, що і cmd.exe. Windows завантажує та виконує програму подібно до інших програм. Такі команди називаються зовнішніми командами (наприклад, MORE або XCOPY).

Розглянемо структуру командного рядка та принцип його роботи. Щоб виконати команду, командний рядок пропонує ввести назву команди (не враховуючи регістр), її аргументи та ключі (якщо потрібно), а потім натиснути клавішу <Enter>.

Наприклад: ***C:\> COPY C:\myfile.txt A:\ /V***

Ім’я команди тут – ***COPY***, параметри – ***C:\myfile.txt*** та ***A:***, а ключ – ***/V***. Звернемо увагу, що в деяких командах ключ може починатися не з символу /, а з символу – (знак мінус), наприклад, ***-V***.

Багато команд Windows мають велику кількість інших аргументів і ключів, які часто важко запам’ятати. Більшість команд мають вбудовану довідку, яка коротко описує мету та синтаксис команди. Можна використовувати ключ ***/?*** для команди, щоб отримати доступ до цієї довідки. Наприклад, якщо запустити команду ***ATTRIB /?***, ми побачимо такий текст:

Відображує або змінює атрибути файлів.

ATTRIB [+R|-R] [+A|-A] [+S|-S] [+H|-H] [[диск:] [шлях] ім’я файлу] [/S]

+ ***Встановлює атрибут.***
- ***Знімає атрибут.***
R Атрибут "Тільки читання".
A Атрибут "Архівний".
S Атрибут "Системний".
H Атрибут "Прихований".
I Атрибут "неіндексованих вміст".
X Атрибут файлу "Без очищення".
V Атрибут цілісності.
[Диск:] [шлях] [ім’я файлу] Вказує файл або набір файлів для обробки.
/S Обробляє файли з зазначеними іменами в поточному каталозі і у всіх його підкаталогах.
/D Обробляє файли і каталоги.
/L Працює з атрибутами самої символьного посилання, а не її цільового об’єкта.

Для деяких команд текст вбудованої довідки може бути достатньо великим для відображення на одному екрані. У цьому випадку можна використовувати команду ***MORE*** та символ конвеєра **|** для послідовного відображення довідки на одному екрані, наприклад:

XCOPY /? | MORE

У цьому випадку після заповнення наступного екрана виведення довідки буде перервано перед натисканням будь-якої клавіші. Крім того, використовуючи вихідні символи переспрямування > та >>, можна відправити відображений текст у текстовий файл для подальшого перегляду. Наприклад, для виведення тексту довідки до команди ***XCOPY*** у текстовому файлі ***xcopy.txt*** слід використати таку команду:

XCOPY /? > XCOPY.TXT

Замість імені файлу можна вказати ім'я пристрою комп'ютера. Windows підтримує такі назви пристроїв: ***PRN*** (принтер), ***LPT1-LPT3*** (відповідний паралельний порт), ***AUX*** (пристрій, підключений до послідовного порту 1), ***COM1-COM3*** (відповідний послідовний порт), ***CON*** (термінал: у разі введення – це клавіатура, у випадку виведення – монітор; за вихідного монітору), ***NUL*** (порожній пристрій, що ігнорує всі операції введення / виведення).

1.1.1. Перенаправлення введення / виведення і конвеєризація (композиція) команд

Розглянемо докладніше UNIX-подібну концепцію перерозподілу стандартних пристроїв вводу-виводу та виконання конвеєра, що підтримується в Windows. Перерозподіляючи пристрої введення-виведення, одна програма може направити свій вихід на вхід іншої програми або використовувати іншу програму як свій вхід для перехоплення виходу іншої програми. Тому можна передавати інформацію між процесами з мінімальними програмними витратами. На практиці це означає, що для програм, що використовують стандартні пристрої введення-виведення, операційна система дозволяє:

- виводити повідомлення програми не на екран (стандартний вихідний потік), а у файл або принтер (перенаправлення виведення);
- зчитувати вхідні дані не з клавіатури (стандартний вхідний потік), а з заздалегідь підготовленого файлу (перенаправлення введення);
- передавати повідомлення, виведені однією програмою, як вхідні дані до іншої програми (конвеєр або композиція команд).

У командному рядку ці функції реалізовані таким чином. Для того, щоб перенаправити текстове повідомлення, яке відображається будь-якою командою, на текстовий файл, вам потрібно використовувати таку конструкцію:

команда > ім'я_файлу

Якщо файл, вказаний для виведення, вже існує, він буде перезаписаний (старий вміст втрачається); якщо він не існує, файл буде створений. Ви також можете не створювати файл знову, але додати інформацію, що відображається командою, у кінець існуючого файлу. Для цього команду перенаправлення виведення потрібно вказати таким чином:

команда >> ім'я_файлу

Використовуючи символ <, ви можете прочитати вхідні дані для цієї команди з клавіатури замість конкретного (заздалегідь підготовленого) файлу:

команда < ім'я_файлу

Ось кілька прикладів перенаправлення введення-виведення.
Вихідні дані вбудованої довідки команди ***COPY*** у файлі ***copy.txt***:
COPY /? > copy.txt

Додавання довідкового тексту команди ***XCOPY*** до файлу ***copy.txt***:
XCOPY /? >> copy.txt

Введення нової дати з файлу ***date.txt*** (***DATE*** – це команда, яка використовується для перегляду та зміни системної дати):
DATE < date.txt

Якщо під час виконання команди виникає помилка, за замовчуванням на екрані відображатиметься повідомлення. Якщо вам потрібні повідомлення про помилки (стандартний потік помилок), ви можете використовувати перенаправлення на текстовий файл:
команда 2> ім'я_файлу

У цьому випадку стандартне виведення буде записано у файл. Також можна відображати довідкові повідомлення та повідомлення про помилки в одному файлі. Робиться це таким чином:
команда > ім'я_файлу 2>&1

Наприклад, у такій команді стандартний вихідний потік та стандартний потік помилок перенаправляються у файл ***copy.txt***:
XCOPY A:\1.txt C:> copy.txt 2>&1

Нарешті, за допомогою конструкції ***команда1 | команда2*** можна використовувати повідомлення, яке відображається першою командою, як вхід для другої команди (конвеєр команд).

Використовуючи механізми перенаправлення введення / виведення та конвеєрну передачу, інформація може надсилатися з командного рядка на різні пристрої та автоматично реагувати на запити команд або програм, що використовують стандартне введення. Командою, придатною для вирішення таких завдань, є ***ECHO [повідомлення]***, яка виводить повідомлення на екран. Розглянемо приклади використання цієї команди.
Надсилання символу запуску на принтер:
ECHO ^L > PRN

Видалення всіх файлів в поточному каталозі без попередження (автоматична позитивна відповідь на запит на видалення):
ECHO y | DEL *.*

Встановленя телефонного з'єднання з командного рядка (модем, підключений до порту COM2):
ECHO ATDT 1 (123) 555-1234 > COM2

1.1.2. Команди MORE і SORT

Однією з найбільш часто використовуваних команд із застосуванням перенаправлення введення / виведення та обробки конвеєра є ***MORE***. Ця команда читає стандартне введення з конвеєра або перенаправленого файлу та видає інформацію частинами, кожна з яких не перевищує розмір екрана. Зазвичай для перегляду довгих файлів використовується ***MORE***. Для цієї команди існує три варіанти синтаксису:

MORE [диск:] [шлях] ім'я_файлу
MORE <[диск:] [шлях] ім'я_файлу
імя_команди | MORE

Параметр ***[диск:] [шлях] ім'я_файлу*** вказує розташування та ім'я файлу, а також кількість переглядів на екрані. Параметр ***ім'я_команди*** визначає команду для відображення її вихідних даних на екрані (наприклад, DIR або команду TYPE, що використовується для відображення вмісту текстового файлу на екрані). Наведемо два приклади.

Щоб переглянути довідку команди ***DIR*** на екрані, потрібно використати таку команду:

DIR /? | MORE

Для поекранного перегляду текстового файлу ***news.txt*** можливі зазначені варіанти команд:

MORE news.txt
MORE < news.txt
TYPE news.txt | MORE

Ще однією поширеною командою, яка використовує перенаправлення введення- виведення та конвеєризацію, є ***SORT***. Ця команда використовується як фільтр: вона зчитує символи у заданому стовпці, сортує їх за зростанням або спаданням та відображає відсортовану інформацію у файлі, екрані чи іншому пристрої. Для цієї команди є два варіанти синтаксису:

SORT [/R] [/+n] [[диск1:] [шлях1] файл1] [> [диск2:] [шлях2] файл2]

або

[Команда] | SORT [/R] [/+n] [> [диск2:] [шлях2] файл2]

У першому випадку параметр ***[диск1:] [шлях1] файл1*** вказує ім'я файлу, що підлягає сортуванню. У другому випадку вихідні дані вказаної команди будуть відсортовані. Якщо параметри ***файл1*** або команда не вказані, ***SORT*** зчитує дані зі стандартного пристрою введення. Параметр ***[диск2:] [шлях2] файл2*** вказує файл, на який

спрямоване відсортоване виведення; якщо цей параметр не вказаний, виведення буде надіслано на стандартний пристрій виведення. За замовчуванням сортування виконується у порядку зростання. Ключ ***/R*** дозволяє змінити порядок сортування (від Z до A, потім від 9 до 0). Наприклад, щоб переглянути файл ***price.txt***, відсортований у зворотному порядку, необхідно вказати таку команду:

SORT /R < price.txt | MORE

Ключ ***/+n*** визначає порядок сортування у файлі за символами в n-му стовпці. Наприклад, ***/+10*** означає, що сортування повинно починатися з 10-ї позиції в кожному рядку. За замовчуванням файли сортуються за одним стовпцем.

1.1.3. Умовне виконання і угруповання команд

У командному рядку Windows ви можете використовувати спеціальні символи, які дозволяють вводити декілька команд одночасно та контролювати роботу команд відповідно до результатів виконання команд. За допомогою цього умовного символу обробки ви можете записати вміст невеликого пакетного файлу в один рядок, а потім виконати складену команду.

Використовуючи символ &, ви можете розділити кілька утиліт на одному командному рядку, і вони працюватимуть одна за одною. Наприклад, якщо ви введете команду DIR & PAUSE & COPY /? і натисните клавішу <Enter>, спочатку відображатиметься вміст поточного каталогу, а потім натиснете будь-яку клавішу – вбудовану довідку команди COPY. Символ ^ дозволяє використовувати символи команд як текст, тобто значення спеціальних символів буде проігноровано. Наприклад, якщо ви вводите: ***ECHO Абв & COPY /?*** і натиснете клавішу ***<Enter>***, то будуть виконані поспіль дві команди: ***ECHO Абв і COPY /?*** (команда ***ECHO*** виведе на екран символи, зазначені в командному рядку після неї). Якщо ж виконати команду ***ECHO Абв ^ & COPY /?***, то на екран буде виведено ***Абв & COPY /?***

У цьому випадку просто буде виконана команда ***ECHO*** з відповідними параметрами.

Умовна обробка команд у Windows здійснюється за допомогою символів && та ||. таким чином. Тільки після того, як команда перед подвійним амперсандом буде успішно виконана, він запустить наступну команду у командній оболонці. Наприклад, якщо у кореневому каталозі диска ***C:*** є файл ***plan.txt***, виконання рядка ***TYPE C:\plan.txt && DIR*** відобразить вміст файлу, а потім вміст поточного каталогу. Якщо файлу ***C:\plan.txt*** не існує, команда ***DIR*** не буде виконана.

Два символи **||** виконують протилежну операцію в командному рядку, тобто команда виконується після цих символів, лише якщо команда після цих символів не виконана успішно. Отже, якщо у попередньому прикладі файлу ***C:\plan.txt*** не існує, вміст поточного каталогу ***DIR*** відображатиметься на екрані через виконання ***TYPE C:\plan.txt || DIR***.

Зверніть увагу, що умовна обробка застосовується лише до найближчої команди, тобто в рядку

TYPE C:\plan.txt && DIR & COPY /?

Команда COPY /? буде виконана незалежно від результату команди ***TYPE C:\plan.txt***.

Кілька утиліт можна згрупувати в дужках у командному рядку. Наприклад, розглянемо наступні два рядки:

TYPE C:\plan.txt && DIR & COPY /?
TYPE C:\plan.txt && (DIR & COPY /?)

У першому з них умовний символ обробки ***&&*** застосовується лише до команди ***DIR***, а у другому – до двох команд одночасно: ***DIR*** та ***COPY***.

1.2. Команди для роботи з файловою системою

Розглянемо деякі найбільш часто використовувані команди роботи з файлами та каталогами. Спочатку звернемо увагу на деякі особливості визначення шляху до файлу в Windows. Нагадаємо, що файлова система представляється деревоподібною структурою, і ім'я файлу вказується у форматі ***[диск:] [шлях\] ім'я_файлу***, тобто обов'язковим параметром є лише ім'я файлу. У цьому випадку, якщо шлях починається з символу «\», маршрут обчислюється з кореневого каталогу, інакше він обчислюється з поточного каталогу. Наприклад, ім'я ***C:\123.txt*** визначає файл ***123.txt*** у поточному каталозі на диску ***C:***, ім'я ***C:\123.txt*** – це файл ***123.txt*** у кореневому каталозі диска ***C:***, а назва ***ABC\123.txt*** – файл ***123.txt*** у підкаталозі ***ABC*** поточного каталогу.

Поточний каталог і батьківський каталог мають спеціальні імена. Поточний каталог представлений символом «**.**», її батьківський каталог – символами «**..**». Наприклад, якщо поточним каталогом є ***C:\WINDOWS***, шлях до файлу ***autoexec.bat*** у кореневому каталозі диска ***C:*** можна записати як ***..\autoexec.bat***.

Ви можете використовувати так звані символи підстановки або шаблони в іменах файлів (але не для дисків або каталогів): «***?***» та «*****». Символ «*****» у назві файлу представляє будь-яку кількість будь-яких дійсних символів, символ «***?***» – довільний символ або його відсутність. Наприклад, імена ***text121.txt*** та ***text911.txt*** застосовуються до шаблону ***text??1.txt***, імена ***text.txt*** та ***textab12.txt*** – до шаблону ***text*.txt***, а всі

файли з назвою ***text*** з будь-яким розширення – до шаблону ***text.**** . Для того, щоб використовувати довгі імена файлів під час роботи у командній оболонці, їх потрібно укласти у подвійні лапки. Наприклад, для запуску файлу з іменем «***MyApplication.exe***» з каталогу «***MyDocuments***», введіть «***C:\MyDocuments\MyApplication.exe***» у командному рядку, а потім натисніть клавішу <Enter>.

Тепер перейдемо безпосередньо до команд роботи з файловою системою.

Команда CD

Ви можете змінити поточний каталог за допомогою такої команди:

CD [диск:] [шлях\]

Враховуючи наведений вище опис, вказується шлях до необхідного каталогу. Наприклад, команда ***CD *** переходить у кореневий каталог поточного диска. Якщо запустити команду ***CD*** без параметрів, на екрані відобразиться назва поточного диска та каталогу.

Команда COPY

Під час роботи на комп’ютері одним із найпоширеніших завдань є копіювання та переміщення файлів з одного місця в інше. Скопіювати один або декілька файлів можливо за допомогою команди ***COPY***.

Синтаксис команди:

COPY [/A | /B] джерело [/A | /B] [+ джерело [/A | /B] [+ ...]] [Результат [/A | /B]] [/V] [/Y | /-Y]

У табл. 1.1 коротко описуються параметри та ключі команди COPY.

Табл. 1.1

Параметри та ключі команди ***COPY***

Ключ	Опис
/A	файл є текстовим файлом ASCII, тобто кінець файлу представлений символом із кодом ***ASCII 26 (<Ctrl> + <Z>)***
/B	файл є бінарним. Цей ключ вказує на те, що інтерпретатор команд повинен зчитувати з джерела кількість байтів, зазначену за розміром у папці скопійованого файлу
результат	каталог, де розміщується результат копіювання та/або ім’я файлу, який потрібно створити
/V	перевірка правильності копії, порівнюючи файли після копіювання
/Y	вимкнення режиму запиту на підтвердження заміни файлу
/-Y	увімкнення режиму запиту на підтвердження заміни файлу

Нижче наведено кілька прикладів використання команди ***COPY***.

1. Копіювання файлу abc.txt з поточного каталогу в каталог ***D:\PROGRAM*** під тим же ім’ям:

COPY abc.txt D:\PROGRAM

2. Копіювання файлу ***abc.txt*** з поточного каталогу в каталог ***D:\PROGRAM*** під новим ім’ям ***def.txt***:

COPY abc.txt D:\PROGRAM\def.txt

3. Копіювання всіх файлів з розширенням ***txt*** з диска ***A:*** у каталог ***‘Мої документи’*** на диску ***C:***

COPY A:*.txt "C:\Мої документи"

Якщо цільовий файл не розміщено в команді, команда COPY створить копію вихідного файлу з тим самим іменем, датою та часом створення, що і вихідний файл, і помістить нову копію в поточний каталог поточного диску. Наприклад, щоб скопіювати всі файли з кореневого каталогу диска ***A:*** у поточний каталог, слід просто запустити таку команду:

COPY A: \ *.*

Як джерело або результат під час копіювання ви можете вказати не тільки ім’я файлу, але й ім’я пристрою комп’ютера. Наприклад, щоб надрукувати файл ***abc.txt*** на принтері, можна скористатися такою командою, щоб скопіювати цей файл на пристрій ***PRN***:

COPY abc.txt PRN

Ще один цікавий приклад: створити новий текстовий файл і записати в нього інформацію без використання текстового редактора. Для цього просто введіть команду ***COPY CON my.txt***, вона скопіює те, що ви введете на клавіатурі, у файл ***my.txt*** (якщо файл існує, його буде перезаписано, інакше він буде створений). Щоб завершити запис, введіть символ кінця файлу, тобто натисніть клавіші ***<Ctrl>*** + ***<Z>***.

Команда ***COPY*** також може об’єднати (склеїти) кілька файлів в один файл. Для цього потрібно вказати один файл результату та кілька вихідних. Цього можна досягти, використовуючи символи підстановки (***?*** та *) або такий формат: ***файл1+файл2+файл3***. Наприклад, для об’єднання файлів ***1.txt*** та ***2.txt*** у файли ***3.txt*** можна вказати таку команду:

COPY 1.txt+2.txt 3.txt

Щоб об’єднати всі файли з розширенням ***dat*** у поточному каталозі в один файл ***all.dat***, виконайте наведені нижче дії: ***COPY /B *.dat all.dat***

Ключ ***/B*** використовується для запобігання усіченню об’єднаних файлів, оскільки у результаті об’єднання файлів команда ***COPY*** за замовчуванням вважає файли текстовими.

Якщо ім’я цільового файлу співпадає з ім’ям одного з копійованих файлів (крім першого файлу), оригінальний вміст цільового файлу буде втрачено. Якщо ім’я цільового файлу опущено, використовується перший файл у списку. Наприклад, команда ***COPY 1.txt+2.txt*** додасть вміст файлу ***2.txt*** до вмісту файлу ***1.txt***. Ви також можете використовувати команду ***COPY***, щоб призначити поточну дату та час файлу, не змінюючи його вміст. Для цього введіть команду:

COPY /B 1.txt+,,

Кома тут означає, що параметр приймача опущений, що призводить до бажаного результату.

Команда ***COPY*** також має свої недоліки. Наприклад, ви можете використовувати її для копіювання прихованих файлів та системних файлів, файлів нульової довжини та файлів у підкаталогах. Крім того, якщо ***COPY*** зустріне файл, який не вдається скопіювати у поточний момент (наприклад, він уже зайнятий іншою програмою), процес копіювання буде повністю перерваний, а інші файли не будуть скопійовані.

Команда XCOPY

Проблеми, згадані в описі команди ***COPY***, можна вирішити за допомогою команди ***XCOPY***, яка надає більше можливостей для копіювання. Однак слід зазначити, що ***XCOPY*** можна використовувати лише з файлами та каталогами, однак не можна з пристроями.

XCOPY джерело [результат] [ключі]

Команда ***XCOPY*** має багато ключів, далі ми розглянемо найбільш важливі з них.

1. Ключ ***/D [:[дата]]*** дозволяє копіювати лише ті файли, які не були змінені до вказаної дати. Якщо параметр дати не вказаний, копіювання буде виконано лише тоді, коли джерело більш нове, ніж результат. Наприклад, команда ***XCOPY "C:\My Documents*.*" "D:\BACKUP\My Documents" /D*** скопіює тільки файли, які були змінені після останньої копії цього типу або файли з каталогу ***"C:\My Documents"***, які не містились у каталозі ***"D:\BACKUP\My Documents"***.
2. Ключ ***/S*** дозволяє копіювати всі непорожні підкаталоги у вихідному каталозі.
3. За допомогою ключа ***/E*** можна скопіювати всі підкаталоги, включаючи порожні.
4. Якщо вказано ключ ***/C***, копіювання продовжиться, навіть якщо сталася помилка. Це корисно для операцій копіювання (таких як резервне копіювання даних), що виконуються з групами файлів.

5. У разі копіювання декількох файлів, де цільовий файл відсутній, важливим є ключ ***/I***. Коли вказаний цей ключ, команда ***XCOPY*** передбачає, що цільовий файл повинен бути каталогом. Наприклад, якщо ви вказали перемикач ***/I*** в команді для копіювання всіх файлів із розширенням ***txt*** з поточного каталогу до неіснуючого підкаталогу ***TEXT***, цей підкаталог буде створений без інших запитів: ***XCOPY *.txt TEXT /I***.

6. Ключі ***/Q***, ***/F*** та ***/L*** відповідають за режим відображення під час копіювання. Коли вказано ключ ***/Q***, під час копіювання ім'я файлу не відображається, а ключ ***/F*** відображає повний шлях до джерела та результату. Ключ ***/L*** означає, що відображаються лише файли, які потрібно скопіювати (копіювання не виконується).

7. Використовуйте ключ ***/H*** для копіювання прихованих файлів та системних файлів, а ключ ***/R*** – для заміни файлів, що призначені лише для читання. Наприклад, щоб скопіювати всі файли з кореневого каталогу диска ***C:*** (включаючи системні та приховані файли) в каталог ***SYS*** диска ***D:***, потрібно ввести таку команду: ***XCOPY C:*.* D:\SYS /H***.

8. Ключ ***/T*** дозволяє використовувати ***XCOPY*** для копіювання лише структури вихідних каталогів, не копіюючи файли в цих каталогах, а також виключаючи порожні каталоги та підкаталоги. Щоб все-таки включати порожні каталоги та підкаталоги, слід використовувати комбінацію ключів ***/T /E***.

9. За допомогою ***XCOPY*** під час копіювання можна оновлювати лише існуючі файли (нові файли не записуються). Для цього використовується ключ ***/U***. Наприклад, якщо каталог ***C:\2*** містить файли ***a.txt*** і ***b.txt***, а каталог ***C:\1*** містить файли ***a.txt***, ***b.txt***, ***c.txt*** і ***d.txt***, команда виконується ***XCOPY C:\1 C:\2 /U***. У каталозі ***C:\2*** досі є лише два файли, ***a.txt*** і ***b.txt***, вміст яких буде замінено вмістом відповідних файлів у каталозі ***C:\1***. Якщо файл, призначений лише для читання, скопійовано за допомогою ***XCOPY***, цей атрибут буде видалений зі скопійованого файлу за замовчуванням. Для того, щоб скопіювати не тільки дані, а й атрибути файлу, потрібно використовувати ключ ***/K***.

10. Ключі ***/Y*** та ***/-Y*** визначають, чи потрібно запитувати підтвердження перед заміною файлу під час копіювання. ***/Y*** – скасувати запит на підтвердження для перезапису існуючого цільового файлу, ***/-Y*** – підтвердити запит на перезапис існуючого цільового файлу.

Команда DIR

Ще однією дуже корисною командою є ***DIR [диск:] [шлях] [ім'я файлу] [ключі]***, що використовується для виведення інформації про вміст дисків та каталогів. Аргумент ***[диск:] [шлях]*** визначає диск і каталог, вміст якого має відображатися. Аргумент ***[ім'я файлу]***

визначає файл або групу файлів, які потрібно включити до списку. Наприклад, під час виконання команди ***DIR C:\ *.bat*** усі файли із розширенням ***bat*** відображатимуться у кореневій директорії диска ***C:***.

Якщо команда ***DIR*** вказана без аргументів, вона вказує мітку диска та його серійний номер, імена файлів та підкаталогів у поточному каталозі (коротка версія та довга версія), а також дату та час останньої модифікації. Після цього кількість файлів у каталозі, загальну кількість файлів, яку він займає (у байтах), і кількість вільного місця на диску. Наприклад:

```
C:\>dir
 Volume in drive C has no label.
 Volume Serial Number is 2806-3E3F

 Directory of C:\

12/08/2018  10:00 AM    <DIR>          $WINDOWS.~BT
12/08/2018  10:32 AM    <DIR>          ESD
07/04/2018  01:42 PM    <DIR>          inetpub
09/12/2016  01:36 PM    <DIR>          Logs
02/19/2021  08:56 PM    <DIR>          Microsoft
07/16/2016  03:23 PM    <DIR>          PerfLogs
03/17/2021  06:39 PM    <DIR>          Program Files
02/19/2021  08:59 PM    <DIR>          Program Files (x86)
12/19/2018  02:27 PM    <DIR>          python
03/05/2021  01:11 PM    <DIR>          Ruby30-x64
04/19/2019  02:03 PM    <DIR>          Users
02/28/2021  07:58 PM    <DIR>          Windows
               0 File(s)              0 bytes
              12 Dir(s)  98,781,413,376 bytes free
```

За допомогою ключів команди ***DIR*** можна задати різні режими розташування, фільтрації і сортування. Під час використання ключа ***/W*** перелік файлів виводиться в широкому форматі з максимально можливим числом імен файлів або каталогів на кожному рядку. Наприклад:

```
C:\>dir /w
 Volume in drive C has no label.
 Volume Serial Number is 2806-3E3F

 Directory of C:\

[$WINDOWS.~BT]         [ESD]                  [inetpub]              [Logs]
[Microsoft]            [PerfLogs]             [Program Files]        [Program Files (x86)]
[python]               [Ruby30-x64]           [Users]                [Windows]
               0 File(s)              0 bytes
              12 Dir(s)  98,781,351,936 bytes free
```

У разі використання ключа ***/A [[:]атрибути]*** будуть відображені тільки каталоги з вказаними атрибутами (***R*** – «лише для читання», ***A*** – «архівний», ***S*** – «системний», ***H*** – «прихований», префікс «**-**» має значення заперечення). Якщо ключ ***/A*** використовується з кількома значеннями атрибутів, відображатимуться лише імена файлів з атрибутами, що відповідають вказаному. Наприклад, щоб відобразити

імена всіх прихованих та системних файлів у кореневому каталозі диска ***C:***, ви можете вказати таку команду: ***DIR C:\ /A:HS***, а щоб вивести всі файли, крім прихованих, слід використати команду ***DIR C:\ /A:-H***.

Зверніть увагу, що атрибут каталогу відповідає букві ***D***, наприклад, щоб відобразити список усіх каталогів на диску ***C:***, вам потрібно вказати таку команду: ***DIR C:\ /A:D***.

Ключ ***/O [[:]сортування]*** визначає порядок сортування, за яким вміст каталогу буде сортуватися відповідно до виведення команди ***DIR***. Якщо цей ключ пропущено, ***DIR*** надрукує імена файлів та каталогів у тому порядку, в якому файли та каталоги містяться в каталозі. Якщо ключ ***/O*** вказаний, а параметр сортування не вказаний, ***DIR*** відображатиме назви в алфавітному порядку. У параметрі сортування можна використовувати такі значення: ***N*** – за іменем (літери), ***S*** – за розміром (починаючи з меншого), ***E*** – за розширенням (в алфавітному порядку), ***D*** – за датою (від найдавнішої), ***A*** – за датою завантаження (також від найдавнішої), ***G*** – почати список з каталогів. Префікс «**-**» позначає зворотний порядок. Якщо вказано кілька значень порядку сортування, файли будуть відсортовані за першою умовою, потім файли будуть відсортовані за другою умовою тощо.

Ключ ***/S*** означає виведення списку файлів із зазначеного каталогу та його підкаталогів. Ключ ***/B*** перелічує лише ім’я каталогу та ім’я файлу (довгий формат) у кожному рядку, включаючи розширення. У той же час виводиться лише основна інформація без остаточної інформації. Наприклад:

```
C:\Program Files\Java\jdk1.8.0_181\bin>dir /b
appletviewer.exe
extcheck.exe
idlj.exe
jabswitch.exe
jar.exe
jarsigner.exe
java-rmi.exe
java.exe
javac.exe
javadoc.exe
javafxpackager.exe
javah.exe
javap.exe
javapackager.exe
javaw.exe
javaws.exe
jcmd.exe
jconsole.exe
jdb.exe
jdeps.exe
jhat.exe
jinfo.exe
jjs.exe
jli.dll
```

Команди MKDIR і RMDIR

Для створення нового каталогу та видалення існуючого порожнього каталогу потрібно використовувати команди ***MKDIR [диск:] шлях*** і ***RMDIR [диск:] шлях [ключі]*** (або їх скорочення MD та RD) відповідно. Наприклад:

MKDIR "D:\Examples"
RMDIR "D:\Examples"

Якщо каталог або файл із зазначеним іменем вже існує, команду ***MKDIR*** неможливо виконати. Якщо каталог, який потрібно видалити, не порожній, команда ***RMDIR*** не буде виконана.

Команда DEL

Ви можете видалити один або кілька файлів за допомогою такої команди:

DEL [диск:] [шлях] ім'я_файлу [ключі].

Можна використовувати символи підстановки ***?*** та ******* для видалення кількох файлів. Ключ ***/S*** дозволяє вам видалити вказаний файл з усіх підкаталогів, ключ ***/F*** – видалити файли, доступні лише для читання, примусово, ключ ***/A [[:]атрибути]*** – вибрати файли, які потрібно видалити, за атрибутами (подібно до аналогічного ключа команди ***DIR***).

Команда REN

За допомогою команди ***REN (RENAME)*** можна перейменовувати файли та каталоги. Синтаксис команди такий:

REN [диск:] [шлях] [каталог1 | файл1] [каталог2 | файл2]

Параметр ***каталог1 | файл1*** тут вказує ім'я каталогу / файлу, який потрібно змінити, а ***каталог2 | файл2*** вказує нове ім'я каталогу / файлу. Можна використовувати символи підстановки ***?*** та ******* в будь-якому аргументі команди ***REN***. Символ, представлений шаблоном у аргументі ***файл2***, буде однаковим із відповідним символом у аргументі ***файл1***. Наприклад, щоб змінити всі файли з розширенням ***txt*** у поточному каталозі розширень на ***doc***, потрібно ввести таку команду:

REN * .txt * .doc

Якщо файл з іменем ***файл2*** вже існує, команда ***REN*** зупинить виконання і виведеться повідомлення про те, що файл вже існує або зайнятий. Крім того, ви можете вказати інший диск або каталог у команді ***REN*** для створення результуючого каталогу та файлів.

У свою чергу для переміщення файлів та каталогів використовується команда ***MOVE***.

Команда MOVE

Синтаксис команди для переміщення одного або декількох файлів:
MOVE [/Y | /-Y] [диск:] [шлях] імя_файла1 [...] результуючий_файл

Синтаксис команди для перейменування папки:
MOVE [/Y | /-Y] [диск:] [шлях] каталог1 каталог2

Тут параметр ***результуючий_файл*** вказує розташування нового файлу і може включати ім'я диска, двокрапку, ім'я каталогу або їх комбінацію. Якщо ви переміщуєте лише один файл, ви можете вказати його нову назву. Це дозволяє негайно перемістити файли. Наприклад:
MOVE "C:\Мої документи\список.txt" D:\list.txt

Якщо вказано ключ ***/-Y***, під час створення каталогу та заміни файлів буде видано запит на підтвердження. Ключ ***/Y*** – надходження такого запиту.

1.3. Мова інтерпретатора cmd.exe. Командні файли

Мова оболонки командного рядка в Windows реалізована як командні (або пакетні) файли. Пакетний файл Windows – це звичайний текстовий файл із розширенням ***bat*** або ***cmd***, який містить діючі команди операційної системи (зовнішні та внутрішні команди) та деякі інші інструкції та ключові слова. Ці команди та ключові слова роблять командні файли схожими з файлами, написаними на алгоритмічних мовах програмування. Наприклад, якщо ви напишете такі команди у файл:

```
C:\
CD %TEMP%
DEL /F *.tmp
```

Запустите його для виконання (подібно виконуваному файлу з розширенням ***com*** або ***exe***), і тоді ми видалимо всі файли з тимчасового каталогу Windows. Отже, виконання командного файлу призводить до того самого результату, що і послідовне введення команд, записаних у ньому. При цьому не виконується попередня компіляція або перевірка синтаксису коду; якщо трапляється неправильна команда, вона ігнорується. Очевидно, що якщо вам доводиться часто виконувати одні й ті самі операції, використання пакетних файлів може заощадити багато часу.

1.3.1. Виведення повідомлень і дублювання команд

За замовчуванням команди пакетного файлу перед виконанням виводяться на екран, що виглядає не дуже естетично. За допомогою команди ***ECHO OFF*** можна вимкнути дублювання команд, що йдуть після неї (сама команда ***ECHO OFF*** при цьому все ж дублюється).

Наприклад:

```
REM Наступні дві команди будуть дублюватися на екрані ...
DIR C:\
ECHO OFF
REM А решта вже не будуть
DIR F:\
```

Виконання файлу з представленим вище кодом призведе до такого результату:

```
F:\>REM ╨э╨░╤Б╤В╤Г╨┐╨╜╤Ц ╨┤╨▓╤Ц ╨║╨╛╨╝╨░╨╜╨┤╨╕ ╨▒╤Г╨┤╤Г╤В╤М ╨┤╤Г╨▒╨╗╤О╨▓╨░╤В╨╕╤Б╤П ╨╜╨░ ╨╡╨║╤А╨░╨╜╤Ц ...

F:\>DIR C:\
 Том в устройстве C не имеет метки.
 Серийный номер тома: 5080-D3A0

 Содержимое папки C:\

20.01.2021  12:16    <DIR>          Microsoft
04.03.2021  21:52    <DIR>          msys64
06.02.2021  13:33    <DIR>          OpenServer
22.01.2021  23:43    <DIR>          PerfLogs
09.04.2021  20:13    <DIR>          Program Files
26.03.2021  22:20    <DIR>          Program Files (x86)
04.03.2021  21:46    <DIR>          Ruby30-x64
25.02.2021  17:16    <DIR>          temp
19.01.2021  22:13    <DIR>          Users
26.03.2021  22:20    <DIR>          Windows
               0 файлов              0 байт
              10 папок  141 592 911 872 байт свободно

F:\>ECHO OFF
 Том в устройстве F имеет метку Новый том
 Серийный номер тома: C207-3E8C

 Содержимое папки F:\

23.11.2016  23:56         4 730 880 BackupUnicode.cbk
02.12.2020  22:25         1 159 154 Conference Program (1).pdf
```

Для відновлення режиму дублювання використовується команда ***ECHO ON***. Крім цього, можна відключити дублювання будь-якого окремого рядка в командному файлі, написавши на початку цього рядка символ @, наприклад:

```
ECHO ON
REM Команда DIR C:\ дублюється на екрані
DIR C:\
REM А команда DIR F:\ - ні
@DIR F:\
```

Результат виконання буде таким:

```
F:\>ECHO ON

F:\>REM Команда DIR C:\ дублюється на екран?

F:\>DIR C:\
 Том в устройстве C не имеет метки.
 Серийный номер тома: 5080-D3A0
```

```
 Содержимое папки C:\

20.01.2021  12:16    <DIR>          Microsoft
04.03.2021  21:52    <DIR>          msys64
06.02.2021  13:33    <DIR>          OpenServer
22.01.2021  23:43    <DIR>          PerfLogs
09.04.2021  20:13    <DIR>          Program Files
26.03.2021  22:20    <DIR>          Program Files (x86)
04.03.2021  21:46    <DIR>          Ruby30-x64
25.02.2021  17:16    <DIR>          temp
19.01.2021  22:13    <DIR>          Users
26.03.2021  22:20    <DIR>          Windows
               0 файлов              0 байт
              10 папок  141 466 046 464 байт свободно

F:\>REM А команда DIR F:\ - н?
 Том в устройстве F имеет метку Новый том
 Серийный номер тома: C207-3E8C

 Содержимое папки F:\

23.11.2016  23:56         4 730 880 BackupUnicode.cbk
```

Таким чином, якщо поставити на початок файлу команду ***@ECHO OFF***, то це вирішить всі проблеми з дублюванням команд.

У пакетному файлі можна виводити на екран рядки з повідомленнями. Робиться це за допомогою команди ***ECHO повідомлення***. Наприклад:

@ECHO OFF
ECHO Привіт!

Команда ***ECHO.*** виводить на екран порожній рядок. Слід звернути увагу, що точка повинна слідувати безпосередньо за словом «***ECHO***». Наприклад:

```
@ECHO OFF
ECHO Привіт!
ECHO.
ECHO Бувай!
```

Виконання такого пакетного файлу призведе до такого результату:

```
F:\>example1.bat
Прив?т!

Бувай!
F:\>
```

Часто зручно використовувати команду ***CLS***, щоб повністю очистити екран для перегляду повідомлень, що виводяться з командного файлу. За допомогою механізму перенаправлення введення/виведення (символи > та >>) повідомлення, що відображається командою ***ECHO***, може бути перенаправлене до певного текстового файлу. Наприклад:

```
@ECHO OFF
ECHO Привіт! > hi.txt
ECHO Бувай! >> hi.txt
```

Наприклад, використовуючи цей метод, ви можете заповнити файл журналу звітами про виконані операції:

```
@ECHO OFF
REM Спроба копіювання
XCOPY C:\PROGRAMS D:\PROGRAMS /s
REM Додавання повідомлення в файл report.txt в разі
REM вдалого завершення копіювання
IF NOT ERRORLEVEL 1 ECHO Успішне копіювання >> report.txt
```

1.3.2. Використання параметрів командного рядка

Під час запуску командного файлу ви можете вказати будь-яку кількість параметрів, які можна використовувати всередині файлу в командному рядку. Наприклад, це дозволяє використовувати один і той же пакетний файл для виконання команд з різними параметрами.

Символи ***%0***, ***%1***, ..., ***%9*** або ***%**** використовуються для доступу до параметрів командного рядка з командних файлів. У цьому випадку змінній %0 буде присвоєне ім’я виконуваного командного файлу, змінним ***%1***, ***%2***, ..., ***%9*** відповідно значення перших дев’яти параметрів командного рядка, а змінній ***%**** – рядок зі всіма аргументами командного рядка. Якщо в командному рядку вказано менше 9 параметрів під час виклику командного файлу, «зайві» змінні в ***%1–%9*** будуть замінені порожніми рядками. Розглянемо такий приклад. Припустимо, існує командний файл ***copier.bat*** із таким вмістом:

```
@ECHO OFF
CLS
ECHO Файл %0 копіює каталог %1 в %2
XCOPY %1 %2 /S
```

Якщо запустити його з командного рядка з двома параметрами, наприклад

copier.bat C:\OpenServer F:\Backup

то на екран виведеться повідомлення

```
Файл example1.bat коп?єю каталог C:\OpenServer в F:\Backup
C:\OpenServer\Open Server.exe
C:\OpenServer\domains\localhost\.htaccess
C:\OpenServer\domains\localhost\bridge.php
C:\OpenServer\domains\localhost\decorator.php
C:\OpenServer\domains\localhost\facade.php
C:\OpenServer\domains\localhost\index.html
C:\OpenServer\domains\localhost\log.txt
C:\OpenServer\domains\localhost\observer.php
C:\OpenServer\install\Install-archive-dlls.exe
C:\OpenServer\install\Install-runtime.exe
```

і здійсниться копіювання каталогу ***C:\OpenServer*** з усіма його підкаталогами в ***F:\Backup***.

За необхідності можна використовувати більше дев'яти аргументів командного рядка. Це досягається за допомогою команди ***SHIFT***, яка змінює значення аргументів з ***%0*** на ***%9*** і копіює кожен аргумент до попереднього аргумента, тобто значення ***%1*** копіюється до значення ***%0***, ***%2*** – у ***%1*** тощо. Значення аргумента командного рядка, що слідує за дев'ятим аргументом, буде присвоєно відповідно у змінну ***%9***. Якщо цей аргумент не вказаний, новим значенням ***%9*** буде порожній рядок. Наприклад, запустимо командний файл ***my.bat*** із командного рядка таким чином:

my.bat p1 p2 p3

Тоді ***%0=my.bat***, ***%1=p1***, ***%2=p2***, ***%3=p3***, а аргументи ***%4–%9*** – це порожні рядки. Після виконання команди ***SHIFT*** значення параметрів будуть змінені таким чином: ***%0=p1***, ***%1=p2***, ***%2=p3***, а аргументи ***%3–%9*** будуть порожніми рядками.

Коли ввімкнено розширену обробку команд, ***SHIFT*** підтримує ключ ***/n***, який визначає початок зміщення параметра на число ***n***, де ***n*** може бути числом від 0 до 9. Наприклад, у команді ***SHIFT /2*** аргумент ***%2*** буде замінено на ***%3***, який у свою чергу буде замінено на ***%4***, а аргументи ***%0*** та ***%1*** залишаться незмінними.

Команда, зворотна ***SHIFT*** (зворотний зсув), відсутня. Після виконання ***SHIFT*** вже не можна відновити параметр (***%0***), який був першим перед зрушенням. Якщо в командному рядку задано більше десяти параметрів, то команду ***SHIFT*** можна використовувати кілька разів.

У командних файлах є деякі можливості синтаксичного аналізу замінних параметрів. Для параметра з номером ***n*** (***%n***) допустимі синтаксичні конструкції (оператори), представлені у табл. 2.2.

При цьому команда, що виконувала б зворотний зсув, відсутня. Після здійснення команди ***SHIFT*** ви більше не зможете відновити перший аргумент (***%0***) перед зміною. Якщо в командному рядку вказано більше десяти параметрів, команду ***SHIFT*** можна використовувати декілька разів.

У командних файлах є деякі можливості аналізу аргументів командного рядка. Для змінної позиційного аргумента (***%n***) дозволена синтаксична структура, яка представлена в табл. 1.2.

Табл. 1.2

Спеціальні змінні для синтаксичного аналізу командного рядка

Конструкція	Опис
%~Fn	Змінна ***%n*** розширюється до повного імені файлу
%~Dn	Із змінної ***%n*** виділяється тільки ім'я диска
%~Pn	Із змінної ***%n*** виділяється тільки шлях до файлу
%~Nn	Із змінної ***%n*** виділяється тільки ім'я файлу
%~Xn	Із змінної ***%n*** виділяється розширення імені файлу
%~Sn	Значення операторів ***N*** та ***X*** для змінної ***%n*** змінюється так, що вони працюють з коротким ім'ям файлу
%~$PATH:n	Проводиться пошук за каталогами, заданими в змінному середовищі PATH, і змінна % n замінюється на повне ім'я першого знайденого файлу

Якщо змінна ***PATH*** не визначена або файл не знайдений у пошуку, ця структура буде замінена порожнім рядком. Звичайно, змінну ***PATH*** тут можна замінити будь-яким іншим дозволеним значенням. Ці граматичні структури можна поєднувати між собою, наприклад: ***%~DPn*** - призначити ім'я диска та шлях зі змінної ***%n***, ***%~NXn*** – ім'я та розширення файлу витягуються із змінної ***%n***.

Припустимо, ми знаходимося в каталозі ***F:\Text*** і запускаємо пакетний файл із аргументом ***Story.doc*** (***%1=Story.doc***). Потім, застосування операторів, описаних у таблиці вище, до параметра ***%1*** дасть такі результати:

%~F1=F:\Text\Story.doc
%~D1=F:
%~P1=\Text
%~N1=Story
%~X1=.doc
%~DP1=F:\Text
%~NX1=Story.doc

1.3.3. Робота зі змінними

Усередині командних файлів можна працювати з так званими змінними середовища (або змінними оточення), кожна з яких зберігається в оперативній пам'яті, має своє унікальне ім'я, а її значенням є рядок. Стандартні змінні середовища автоматично не започатковано в процесі завантаження операційної системи. Такими змінними є, наприклад, ***WINDIR***, яка визначає розташування каталогу Windows, ***TEMP***, яка визначає шлях до каталогу для зберігання тимчасових файлів Windows або ***PATH***, в якій зберігається системний шлях (шлях пошуку), тобто список каталогів, в яких система повинна шукати виконувані файли або файли спільного доступу (наприклад, динамічні бібліотеки). Крім того, в командних файлах за допомогою команди ***SET*** можна оголошувати власні змінні середовища.

У пакетних файлах ви можете використовувати так звані змінні середовища (або змінні оточення), кожна з яких зберігається в оперативній пам'яті, має своє унікальне ім'я, а її значенням є рядок. Стандартні змінні середовища не активуються автоматично під час завантаження операційної системи. Такими змінними є, наприклад, ***WINDIR***, яка визначає розташування каталогу Windows, ***TEMP***, яка визначає шлях до каталогу, що використовується для зберігання тимчасових файлів Windows, або ***PATH***, яка зберігає системний шлях (шлях пошуку), список каталогів, в яких система повинна шукати виконувані файли або файли спільного доступу (наприклад, динамічні бібліотеки). Ви також можете використовувати команду ***SET*** для оголошення власних змінних середовища у командному файлі.

Отримання значення змінної

Для отримання значення певної змінної середовища потрібно ім'я цієї змінної укласти в символи ***%***. Наприклад:

```
@ECHO OFF
CLS
REM Створення змінної MyVar
SET MyVar=Привіт
REM Зміна змінної
SET MyVar=%MyVar%!
ECHO Значення змінної MyVar: %MyVar%
REM Видалення змінної MyVar
SET MyVar=
ECHO Значення змінної WinDir: %WinDir%
```

Із запуском такого командного файлу на екран виведеться рядок:

Значення змінної MyVar: Привіт!

Значення змінної WinDir: C:\WINDOWS

Перетворення змінних як рядків

Ви можете маніпулювати змінними середовища в командних файлах. По-перше, ви можете виконувати операції конкатенації. Для цього просто слід написати поруч значення змінних, що потрібно з'єднати. Наприклад,

```
SET A=Раз
SET B=Два
SET C=%A%%B%
```

Після виконання цих команд у файлі значенням змінної ***C*** буде рядок ***"раздва"***. Не слід використовувати символ + для конкатенації, оскільки він буде просто розглядатися як символ. Наприклад, після запуску файлу з таким вмістом

```
SET A=Раз
SET B=Два
SET C=A+B
ECHO Змінна C=%C%
SET D=%A%+%B%
ECHO Змінна D=%D%
```

на екран виведуться два рядки:

Змінна C=A+B
Змінна D=Раз+Два

По-друге, ви можете використовувати конструкцію ***%ім'я_змінної:~n1,n2%*** для виділення підрядка зі змінних середовища, де число ***n1*** визначає зміщення, відповідне від початку (якщо ***n1*** позитивне) або від кінця (якщо ***n1*** негативне), а число ***n2*** – це кількість вибраних символів (якщо ***n2*** позитивне) або число останнього символу у змінній, який не включений у вибраний підрядок (якщо ***n2*** негативне). Якщо вказано лише один негативний параметр ***-n***, останні ***n*** символів буде видалено. Наприклад, якщо рядок ***"14.04.2021"*** зберігається у змінній ***%DATE%*** (символічне представлення поточної дати за відповідних налаштувань локалізації), після виконання такої команди

```
SET dd1=%DATE:~0,2%
SET dd2=%DATE:~0,-8%
SET mm=%DATE:~-7,2%
SET yyyy=%DATE:~-4%
```

нові змінні матимуть такі значення:

dd1%=21,
dd2%=21,
mm%=04,
%yyyy%=2021.

По-третє, можна виконувати процедуру заміни підрядків за допомогою конструкції ***%ім'я_змінної:s1=s2%*** (в результаті буде повернений рядок, в якій кожне входження підрядка ***s1*** у відповідній змінної середовища замінено на ***s2***). Наприклад, після виконання команд

```
SET a=123456
SET b=%a:23=99%
```

у змінній ***b*** буде зберігатися рядок ***"199456"***. Якщо параметр ***s2*** не вказано, то підрядок ***s1*** буде видалений з рядка, що виводиться, тобто після виконання команди

```
SET a=123456
SET b=%a:23=%
```

у змінній ***b*** буде зберігатися рядок ***"1456"***.

Операції зі змінними як з числами

Увімкнувши розширену обробку команд (Windows використовує цей режим за замовчуванням), ви можете обробляти значення змінних середовища як числа і виконувати арифметичні обчислення щодо них. Для цього слід використати команду ***SET*** із ключем ***/A***. Нижче представлено приклад командного файлу ***add.bat***, який складається з двох чисел, зазначених як аргументи командного рядка, і відображає на екрані їх суму:

```
@ECHO OFF
REM В змінної M буде зберігатися сума
SET /A M=%1+%2
ECHO Сума %1 і %2 дорівнює %M%
REM Вилучимо змінну M
SET M =
```

Локальні зміни змінних

Після закриття файлу всі зміни, внесені до змінних середовища в пакетному файлі за допомогою команди ***SET***, будуть збережені, але вони будуть діяти лише у поточному вікні. Також можна локалізувати зміни змінних середовища в пакетному файлі, тобто автоматично відновити значення всіх змінних у формі перед початком файлу. Для цього використовуються дві команди: ***SETLOCAL*** та ***ENDLOCAL***. Команда ***SETLOCAL*** задає початок області налаштування змінної локального середовища. Іншими словами, екологічні зміни, внесені після ***SETLOCAL***, будуть локальними для поточного пакетного файлу. Кожна команда ***SETLOCAL*** повинна мати відповідну команду ***ENDLOCAL*** для відновлення значення змінної середовища. Зміни, внесені в середовище після команди ***ENDLOCAL***, більше не є локальними для поточного пакетного файлу; їх попередні значення не будуть відновлені після виконання файлу.

1.3.4. Призупинення виконання та виклик командних файлів

Для того, щоб вручну перервати виконання запущеного bat-файлу, потрібно натиснути клавіші <*Ctrl*>+<*C*> або <*Ctrl*>+<*Break*>. Однак часто буває необхідно програмно призупинити виконання командного файлу в певному рядку з видачею запиту на натискання будь-якої клавіші. Це робиться за допомогою команди ***PAUSE***. Перед запуском цієї команди корисно за допомогою команди ***ECHO*** інформувати користувача про дії, які він повинен зробити. Наприклад:

```
ECHO Вставте дискету в дисковод A: і натисніть будь-яку клавішу
PAUSE
```

Команду ***PAUSE*** обов'язково потрібно використовувати під час виконання потенційно небезпечних дій (видалення файлів, форматування дисків та ін.). Наприклад:

```
ECHO Зараз будуть видалені всі файли в C:\Мої документи!
ECHO Для скасування натисніть Ctrl-C
PAUSE
DEL "C:\Мої документи\*.*"
```

З одного командного файлу можна викликати інший, просто вказавши його ім'я. Наприклад:

```
@ECHO OFF
CLS
REM Виведення списку log-файлів
DIR C:\*.Log
REM Передача виконання файлу f.bat
f.bat
COPY A:\*.* C:\
PAUSE
```

Однак у цьому випадку після виконання викликаного файлу управління у файл, викликається, не передається, тобто в наведеному прикладі команда ***COPY A:*.* C:*** (всі наступні за нею команди) ніколи не буде виконана.

Для того, щоб викликати зовнішній командний файл з наступним поверненням в початковий файл, потрібно використовувати спеціальну команду ***CALL файл***.

Наприклад:

```
@ECHO OFF
CLS
REM Виведення списку log-файлів
DIR C:\*.Log
REM Передача виконання файлу f.bat
CALL f.bat
COPY A:\*.* C:\
PAUSE
```

У цьому випадку після завершення роботи файлу ***f.bat*** управління повернеться в первинний файл на рядок, наступний за командою ***CALL*** (в нашому прикладі це команда ***COPY A:*.* C:***).

1.4. Оператори переходу у командних файлах

Пакетний файл може містити мітки та команди ***GOTO*** для переходу за цими мітками. В обробці пакетних файлів будь-який рядок, що починається з двокрапки, вважається міткою. Її назва визначається набором символів після двокрапки до початку або кінця рядка.

Як приклад розглянемо пакетний файл із таким вмістом:

```
@ECHO OFF
COPY %1 %2
GOTO Label1
ECHO Цей рядок ніколи не виконається
:Label1
REM Продовження виконання
DIR %2
```

Дійшовши у файлі до команди ***GOTO Label1***, його виконання буде продовжено з рядка ***REM Продовження виконання***.

У команді переходу у файлі ***GOTO*** можна вказати рядок ***:EOF*** як мітку переходу, яка передає керування в кінець поточного пакетного файлу (це дозволяє легко вийти з командного файлу, не вказуючи в кінці жодних міток). На додаток до команди ***GOTO***, ви також можете перейти до мітки в поточному пакетному файлі. Ви можете використовувати наведену вище команду ***CALL***:

CALL: мітка_аргументу

Коли ви викликаєте таку команду, буде створений новий контекст поточного пакетного файлу із зазначеними параметрами, а керування буде передано інструкції, яка слідує за міткою. Щоб вийти з такого командного файлу, потрібно двічі дійти до кінця. Перший вихід повертає керування інструкцією відразу після рядка ***CALL***, а другий вихід припиняє виконання командного файлу. Наприклад, якщо ви використовуєте параметр ***"Копія-1"*** для запуску командного файлу:

```
@ECHO OFF
ECHO% 1
CALL :2 Копія-2
:2
ECHO% 1
```

то на екран буде виведено три рядки:

```
F:\>example1.bat Копія-1
Копія-1
Коп?я-2
Копія-1
F:\>_
```

Отже, це використання команди ***CALL*** дуже схоже на звичайні виклики підпрограм (процедур) в алгоритмічних мовах програмування.

1.4.1. Оператори умови у командних файлах

Ви можете використовувати команду ***IF ... ELSE*** (може бути відсутнім ключове слово ***ELSE***) для обробки декількох умов у пакетному файлі. У цьому випадку, якщо умова, вказана після ***IF***, приймає справжнє значення, система виконає наступну команду (або кілька команд у дужках), інакше буде виконана команда після ключового слова ***ELSE*** (або кілька команд у дужках).

Перевірка значення змінної

Перший тип умови зазвичай використовується для перевірки значення змінної. Для цього використовуйте два варіанти синтаксису команди ***IF***:

IF [NOT] рядок1==рядок2 команда1 [ELSE команда2]

(квадратні дужки означають, що параметри, котрі містяться в ньому, є необов'язковими) або:

IF [/I] [NOT] рядок1 оператор_порівняння рядок2 команда

Спочатку розглянемо перший варіант. У тому випадку, коли два рядки повністю перекриваються, умова ***рядок1==рядок2*** (два рівні символи повинні бути написані саме тут) вважається дійсною. Параметр ***NOT*** вказує, що вказана команда буде виконана лише тоді, коли порівняні рядки не збігаються.

Рядок може бути літеральним або представляти значення змінної (наприклад, ***%1*** або ***%TEMP%***). Лапки для літеральних рядків не потрібні, наприклад:

IF %1==%2 ECHO Параметри збігаються!
IF %1==Петро ECHO Привіт, Петро!

Зауважимо, що слід бути обережним у порівнянні рядків, зазначених змінними. Справа в тому, що значення змінної може бути порожнім рядком, і тоді командний файл може не виконатись. Наприклад, якщо змінна MyVar не вказана за допомогою команди SET, а файл має оператор умовного типу:

IF %MyVar%==C:\ ECHO Ура!!!

Тоді у процесі виконання значення змінної ***%MyVar%*** замінить порожній рядок і виникне синтаксична помилка. Якщо один із рядків, що підлягає порівнянню, є значенням параметра командного рядка, може статися така сама ситуація, оскільки цей параметр не може бути вказаний під час запуску командного файлу. Тому, порівнюючи рядки, необхідно на початку приписувати їм будь-які символи, наприклад:

IF -%MyVar%==-C:\ECHO Ура!!!

Використовуючи команди IF та SHIFT, ви можете переглядати всі параметри командного рядка файлу, навіть не знаючи їх номерів заздалегідь. Наприклад, наступний командний файл (ми називаємо його ***example1.bat***) показує ім’я запущеного файлу та всі параметри командного рядка:

```
@ECHO OFF
ECHO Виконується файл: %0
ECHO.
ECHO Файл запущений з наступними аргументами ...
REM Початок циклу
:BegLoop
IF -%1==- GOTO ExitLoop
ECHO %1
REM Зрушення параметрів
SHIFT
REM Перехід на початок циклу
GOTO BegLoop
: ExitLoop
REM Вихід з циклу
ECHO.
ECHO Все.
```

Якщо запустити файл ***example1.bat*** з п’ятьма параметрами, то в результаті його виконання на екран буде виведена така інформація:

```
F:\>example1.bat a b c d e
Виконується файл: example1.bat

Файл запущений з наступними аргументами ...
a
b
c
d
e
```

Тепер розглянемо оператор ***IF*** таким чином:

IF [/ I] рядок1 оператор_порівняння рядок2 команда

Синтаксис та значення операторів порівняння наведені в табл. 1.3.

Табл. 1.3

Оператори порівняння

Оператор	**Значення**
EQL	=
NEQ	≠
LSS	<
LEQ	≤
GTR	>
GEQ	≥

Наведемо приклад використання операторів порівняння:

```
@ECHO OFF
CLS
IF -%1 EQL -Василь ECHO Привіт, Василь!
IF -%1 NEQ -Василь ECHO Привіт, але ви не Василь!
```

Ключ ***/I*** (якщо вказано) визначає текстовий рядок, що не враховує регістр. Він також може використовуватися у формі ***рядок1==рядок2*** команди ***IF***. Наприклад, умова ***IF /I DOS==dos ...*** буде правдою.

Перевірка існування заданого файлу

Другий спосіб використання команди ***IF*** – перевірити наявність заданого файлу. Синтаксис для цього випадку:

IF [NOT] EXIST файл команда1 [ELSE команда2]

Якщо вказаний файл існує, умова вважається дійсною. Для імені файлу не потрібні лапки. Наведемо приклад командного файлу, де команда ***IF*** перевіряє, чи існує файл, що вказаний як параметр командного рядка.

```
@ECHO OFF
IF -%1==- GOTO NoFileSpecified
IF NOT EXIST %1 GOTO FileNotExist
REM Виведення повідомлення про знайдений файлі
ECHO Файл '%1' успішно знайдений.
GOTO :EOF

:NoFileSpecified
REM Файл запущений без аргументів
ECHO У командному рядку не вказано ім'я файлу.
GOTO :EOF

:FileNotExist
REM Аргумент командного рядка заданий, але файл не знайдений
ECHO Файл '%1' не знайдений.
```

Перевірка наявності змінної середовища

Подібно до файлів, команда IF дозволяє перевірити, чи існують у системі певні змінні середовища:

IF DEFINED змінна команда1 [ELSE команда2]

Тут умова ***DEFINED*** подібна до умови ***EXISTS*** для існування цього файлу, але приймає за параметр ім’я змінної середовища. Якщо ця змінна визначена, вона повертає істинне значення.

Наприклад:

```
@ECHO OFF
CLS
IF DEFINED MyVar GOTO: VarExists
ECHO Змінна MyVar не визначена
GOTO :EOF

: VarExists
ECHO Змінна MyVar визначена,
ECHO її значення дорівнює %MyVar%
```

Перевірка коду завершення попередньої команди

Іншим способом використання команди ***IF*** є перевірка коду завершення (коду виходу) попередньої команди. У цьому випадку синтаксис ***IF*** такий:

IF [NOT] ERRORLEVEL число команда1 [ELSE команда2]

Тут, якщо остання команда виконання або програма закінчується кодом повернення, рівним або більшим за вказане число, умова вважається дійсною. Наприклад, давайте створимо пакетний файл, який копіює файл ***my.txt*** на диск ***C:*** без відображення повідомлення про копію та попередження у разі помилок:

```
@ECHO OFF
XCOPY my.txt C: \> NUL
REM Перевірка коду завершення копіювання
IF ERRORLEVEL 1 GOTO ErrOccurred
ECHO Копіювання виконано без помилок.
GOTO: EOF

: ErrOccurred
ECHO При виконанні команди XCOPY виникла помилка!
```

В операторі ***IF ERRORLEVEL ...*** ви також можете використовувати числові оператори порівняння, перелічені у таблиці вище:

IF ERRORLEVEL LEQ 1 GOTO Case1

Іноді використання змінної ***%ERRORLEVEL%*** може бути зручнішим для використання коду завершення програми (рядкове представлення поточного значення коду помилки ***ERRORLEVEL***).

Перевірка версії реалізації розширеної обробки команд

Нарешті, для визначення внутрішнього номера версії поточної реалізації розширеної обробки команд, ви можете використовувати оператор ***IF***, як показано нижче:

IF CMDEXTVERSION число команда1 [ELSE команда2]

Тут умова ***CMDEXTVERSION*** застосовується аналогічно умові ***ERRORLEVEL***, але число порівнюється із внутрішнім номером версії. ***Перша*** версія має номер ***1***. Щоразу, коли ви додаєте важливі функції для розширеної обробки команд, номер версії збільшується на одиницю. Якщо розширену обробку команд вимкнено, умова ***CMDEXTVERSION*** ніколи не буде істинною.

1.4.2. Організація циклів у командних файлах

Пакетні файли використовують кілька типів операторів ***FOR*** для організації циклів, які забезпечують такі функції:

– виконати задану команду на всіх елементах зазначеного набору;
– виконати вказану команду для всіх пов'язаних імен файлів;
– виконати вказану команду над усіма відповідними іменами каталогів;
– виконати задану команду для певного каталогу та всіх його підкаталогів;
– отримати послідовність чисел із заданим початком, кінцем та кроком збільшення;
– зчитати та обробити рядки із текстових файлів;
– обробити вихідні рядки для конкретних команд.

Цикл FOR ... IN ... DO ...

Найпростіша версія синтаксису команди ***FOR*** командного файлу така:

FOR %%змінна IN (безліч)
DO команда [параметри]

Увага! Імені змінної повинні передувати два знаки відсотка (*%%*), а не знак відсотка, як і у використанні команди ***FOR*** безпосередньо з командного рядка.

Маємо приклад. Якщо рядок вказаний у пакетному файлі:

```
@ECHO OFF
FOR %%i IN (Раз, Два, Три) DO ECHO %%i
```

, то в результаті його виконання на екрані буде надруковано таке:

```
Раз
Два
Три
```

Параметр ***множина*** у команді ***FOR*** вказує один або кілька рядків тексту, розділеного комами, який буде оброблений зазначеною командою. Тут потрібні дужки. Параметр ***команда [параметри]*** визначає команду, що виконується для кожного елемента набору, і не дозволяє команді ***FOR*** вкладатись у рядок. Якщо в частині рядка використовується кома, значення рядка повинно бути в лапках. Наприклад, через виконання командного файлу

```
@ECHO OFF
FOR %%i IN ("Раз, Два", Три) DO ECHO %%i
```

, на екрані відобразиться:

```
"Раз, Два"
Три
```

Параметр ***%%змінна*** дозволяє підставляти змінну (лічильник циклу), і тут можна використовувати лише імена змінних, що складаються з однієї літери. Після виконання команда ***FOR*** замінить змінну, яка підставляється текстом кожного рядка у зазначеному наборі, поки команда після ключового слова ***DO*** не обробить усі такі рядки.

Щоб уникнути плутанини з параметрами ***%0–%9*** пакетних файлів, використовуйте будь-які символи, крім ***0–9***, для змінних.

Параметр ***множина*** у команді ***FOR*** також може представляти одну або кілька груп файлів. Наприклад, щоб перерахувати всі файли з розширеннями ***txt*** та ***prn*** у каталозі ***C: \TEXT*** без використання команди ***DIR***, ви можете використовувати пакетний файл із таким вмістом:

```
@ECHO OFF
FOR %%f IN (C:\TEXT\*.txt C:\TEXT\*.prn) DO ECHO %%f >> list.txt
```

Коли ви використовуєте команду ***FOR***, обробка триватиме до тих пір, поки всі файли (або групи файлів), зазначені в колекції, не будуть оброблені.

Цикл FOR /D ... IN ... DO ...

Наступні варіанти команди ***FOR*** можуть бути реалізовані за допомогою ключа ***/D***:

FOR /D %%змінна IN (набір) DO команда [параметри]

Якщо набір містить символи підстановки, команда виконується для всіх відповідних назв каталогів замість імен файлів. Припустимо, ми маємо такий командний файл:

```
@ECHO OFF
CLS
FOR /D %%f IN (C:\*.*) DO ECHO %%f
```

ми отримаємо список всіх каталогів на диску C:, наприклад:

```
C:\Microsoft
C:\msys64
C:\OpenServer
C:\PerfLogs
C:\Program Files
C:\Program Files (x86)
C:\Ruby30-x64
C:\temp
C:\Users
C:\Windows
```

Цикл FOR /R ... IN ... DO ...

Використовуйте ключ ***/R***, щоб вказати рекурсію в команді ***FOR***:

FOR /R [[диск:] шлях] %%змінна IN (набір)
DO команда [параметри]

У цьому випадку вказана команда буде виконана для каталогу ***[диск:] шлях*** та у всіх його підкаталогах. Якщо ім'я каталогу не вказано після ключа ***/R***, команда буде виконана з поточного каталогу. Наприклад, ви можете використовувати такий пакетний файл для друку всіх файлів із розширенням ***txt*** у поточному каталозі та всіх його підкаталогах:

```
@ECHO OFF
CLS
FOR /R %%f IN (*.txt) DO PRINT %%f
```

Якщо ви вказали лише крапку (**.**) замість набору, команда перевірить усі підкаталоги поточного каталогу. Наприклад, якщо ми знаходимося в каталозі ***C:\TEXT***, з двома підкаталогами ***BOOKS*** і ***ARTICLES***, то в результаті виконання такого файлу

```
@ECHO OFF
CLS
FOR /R %%f IN (.) DO ECHO %%f
```

на екрані відображатимуться три рядки:

C:\TEXT\.
C:\TEXT\BOOKS\.
C:\TEXT\ARTICLES\.

Цикл FOR / L ... IN ... DO ...

Ключ ***/L*** дозволяє використовувати команду ***FOR*** для реалізації арифметичних циклів. У цьому випадку синтаксис такий:

FOR /L %%змінна IN (початок, крок, кінець) DO команда [параметри]

Тут трійки, зазначені після ключового слова ***IN***, ***(початок, крок, кінець)***, відображаються як послідовність чисел із зазначеним початком, кінцем та приростом. Отже, набір ***(1,1,5)*** відображається в ***(1 2 3 4 5)***, а набір ***(5,-1,1)*** замінюється на ***(5 4 3 2 1)***. Наприклад, в результаті виконання такого пакетного файлу

```
@ECHO OFF
CLS
FOR /L %%f IN (1,1,5) DO ECHO %%f
```

змінна циклу ***%%f*** на відповідних ітераціях матиме значення від 1 до 5, і на екрані буде надруковано п’ять чисел:

```
1
2
3
4
5
```

Числа, отримані у результаті виконання циклу ***FOR /L***, можна використовувати для арифметичних обчислень. Розглянемо командний файл:

```
@ECHO OFF
CLS
FOR /L %%f IN (1,1,5) DO CALL :2 %%f
GOTO: EOF
:2
SET /A M=10*%1
ECHO 10*%1=%M%
```

У третьому рядку в циклі відбувається виклик нового контексту цього файлу з поточним значенням змінної циклу ***%%f*** як параметра командного рядка, причому управління передається на мітку ***:2***. У шостому рядку змінна циклу множиться на десять, і результат записується в змінну ***M***. Таким чином, у результаті виконання цього файлу виведеться така інформація:

```
10*1=10
10*2=20
10*3=30
10*4=40
10*5=50
```

Цикл FOR / F ... IN ... DO ...

Найпотужніші можливості (а також найбільш заплутаний синтаксис) має команда: ***FOR*** з ключем ***/F***:

FOR /F ["ключі"] %%змінна IN (набір)
DO команда [параметри]

Тут параметр ***набір*** містить імена одного або декількох файлів, які по черзі відкриваються, читаються та обробляються. Обробка полягає в тому, щоб прочитати файл, розділити його на окремі текстові рядки, а потім вибрати задану кількість підрядків з кожного рядка. Потім, коли тіло циклу (задана команда) виконується, знайдений підрядок буде використаний як значення змінної.

За замовчуванням ключ ***/F*** витягує перше слово з кожного рядка файлу, очищуючи його від пробілів. Пусті рядки у файлі будуть пропущені. Необов'язковий параметр ***"ключі"*** використовується для заміни правил обробки рядків за замовчуванням. Клавіші укладені в лапки та включають перелічені в табл. 1.4 ключові слова:

Табл. 1.4

Ключові слова для використання циклу ***FOR*** з ключем ***/F***

Ключ	Опис
EOL=C	Визначення символу коментарів на початку рядка (допускається завдання тільки одного символу)
SKIP=N	Число пропускається під час обробки рядків на початку файлу
DELIMS=XXX	Визначення набору роздільників для заміни заданих за замовчуванням пробілу і знака табуляції
TOKENS=X,Y, M-N	Визначення номерів підрядків, що виділяються з кожного рядка файлу і переданих для виконання в тіло циклу

Коли використовується ключ ***TOKENS=X,Y,M-N***, будуть створені інші змінні. Формат ***M-N*** – це діапазон з підрядків, що мають порядкові номери від ***M*** до ***N***. Якщо останнім символом у рядку ***TOKENS*** є зірочка, буде створена додаткова змінна, значенням якої буде весь залишковий текст у рядку після обробки останнього підрядка.

Давайте розглянемо застосування цієї команди на прикладі командного файлу, який аналізує файл ***myfile.txt***:

```
@ECHO OFF
IF NOT EXIST myfile.txt GOTO :NoFile

FOR /F "EOL=; TOKENS=2,3* DELIMS=, " %%i IN (myfile.txt) DO (
  @ECHO %%i %%j %%k
  )

GOTO :EOF
:NoFile
ECHO Не вдалося знайти файл myfile.txt!
```

Тут у другому рядку проводиться перевірка наявності файлу ***myfile.txt***; у разі відсутності цього файлу виводиться попередження. Команда ***FOR*** в третьому рядку обробляє файл ***myfile.txt*** таким чином.

Пропускаються всі рядки, які починаються з крапки з комою ***(EOL =;)***.

Другий і третій підрядки з кожного рядка передаються в тіло циклу, причому підрядки розділяються проміжками (за замовчуванням) і / або комами ***(DELIMS =,)***.

У тілі циклу змінна ***%%i*** використовується для другого підрядка, ***%%j*** – для третього, а ***%%k*** отримує всю решту підрядків після третього.

У нашому прикладі змінна ***%%i*** явно описана в інструкції ***FOR***, а змінні ***%%j*** і ***%%k*** описуються неявно за допомогою ключа ***TOKENS=***. Наприклад, якщо у файлі ***myfile.txt*** були записані такі три рядки:

aaa bbbb cccc, ddddd eeee
fffff, gggg hhhh
; iiii jjjjj kkkkk

то в результаті виконання пакетного файлу parser.bat на екран виведеться:

```
bbbb cccc ddddd eeee
gggg hhhh
```

Ключ ***TOKENS=*** дозволяє витягти з одного рядка файлу до 26 підрядків, тому заборонено використовувати імена змінних, починаються не з букв англійського алфавіту (***a-z***). Слід пам’ятати, що імена змінних ***FOR*** є глобальними, тому одночасно не може бути активно понад 26 змінних.

Команда ***FOR /F*** також дозволяє обробити окремий рядок. Для цього слід ввести потрібний рядок в лапках замість набору імен файлів у дужках. Рядок буде оброблений так, як ніби він взятий з файлу. Наприклад, файл такого змісту:

```
@ECHO OFF
FOR /F "EOL=; TOKENS=2,3* DELIMS=, " %%i IN ("aaa bbbb cccc, ddddd eeee") DO (
  @ECHO %%i %%j %%k
)
```

при своєму виконанні надрукує

```
bbbb cccc ddddd eeee
```

Ви можете використовувати змінні середовища замість явних завдань рядків для розбору, наприклад:

```
@ECHO OFF
SET M = aaa bbb cccc, ddddd eeee
FOR /F "EOL=; TOKENS=2,3* DELIMS=, " %%i IN ("%M%") DO (
    @ECHO %%i %%j %%k
)
```

Нарешті, команда ***FOR /F*** дозволяє обробити вихідний рядок іншої команди. Для цього, замість того, щоб вводити набір імен файлів у дужках, введіть команду в одинарних лапках. Рядок буде передано інтерпретатору команд ***cmd.exe***, а вихідні дані цієї команди будуть записані в пам'ять та оброблені так, ніби вихідний рядок отримано з файлу. Наприклад, такий командний файл:

```
@ECHO OFF
CLS
ECHO Імена змінних середовища:
ECHO.
FOR /F "DELIMS==" %%i IN ('SET') DO ECHO %%i
```

виведе перелік імен всіх змінних середовища, визначених у конкретний час у системі:

```
?мена зм?нних середовища:

ALLUSERSPROFILE
APPDATA
CommonProgramFiles
CommonProgramFiles(x86)
CommonProgramW6432
COMPUTERNAME
ComSpec
FPS_BROWSER_APP_PROFILE_STRING
FPS_BROWSER_USER_PROFILE_STRING
HOMEDRIVE
HOMEPATH
IntelliJ IDEA Educational Edition
LOCALAPPDATA
LOGONSERVER
M
NUMBER_OF_PROCESSORS
OneDrive
OS
Path
PATHEXT
PROCESSOR_ARCHITECTURE
PROCESSOR_IDENTIFIER
PROCESSOR_LEVEL
PROCESSOR_REVISION
ProgramData
ProgramFiles
ProgramFiles(x86)
ProgramW6432
```

У циклі ***FOR*** допускається застосування тих самих синтаксичних конструкцій (операторів), що і для замінних параметрів (табл. 1.5).

Табл. 1.5

Синтаксичні конструкції, що можуть використані у циклі

Оператори	Опис
%~Fi	Змінна ***%i*** розширюється до повного імені файлу
%~Di	Із змінної ***%i*** виділяється тільки ім'я диска
%~Pi	Із змінної ***%i*** виділяється тільки шлях до файлу
%~Ni	Із змінної ***%i*** виділяється тільки ім'я файлу
%~Xi	Із змінної ***%i*** виділяється розширення імені файлу
%~Si	Значення операторів N та X для змінної ***%i*** змінюється так, що вони працюють з коротким ім'ям файлу

Якщо ви маєте намір використовувати розширення підстановки значень у команді FOR, слід обережно вибирати імена змінних, щоб вони не перетиналися з позначеннями формату. Наприклад, якщо ми знаходимося в каталозі ***C:\Ruby30-x64\bin\ ruby_builtin_dlls*** і запускаємо пакетний файл із таким вмістом:

```
@ECHO OFF
CLS
FOR %%i IN (*.dll) DO ECHO %%~Fi
```

то на екран виведуться повні імена всіх файлів з розширенням ***dll***:

```
C:\Ruby30-x64\bin\ruby_builtin_dlls\libcrypto-1_1-x64.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libffi-7.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libgcc_s_seh-1.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libgdbm-6.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libgdbm_compat-4.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libgmp-10.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libiconv-2.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libintl-8.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libssl-1_1-x64.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libssp-0.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libwinpthread-1.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\libyaml-0-2.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls\zlib1.dll
C:\Ruby30-x64\bin\ruby_builtin_dlls>
```

Розділ 2.
Сервер сценаріїв Windows script host

У попередньому розділі розглядалася мова командних файлів як інструмент автоматизації, що підтримується у всіх версіях Windows. Однак важко писати будь-які складні сценарії за допомогою оболонки ***cmd.exe***: без зрілої інтерактивності її не можна використовувати безпосередньо з робочим столом Windows та реєстром.

Щоб вирішити цю проблему, Microsoft розробила сервер сценаріїв ***Windows Script Host (WSH)***, який дозволяє запускати сценарії, написані будь-якою мовою за умови встановлення для неї відповідного модуля, що підтримує технологію ActiveX Scripting. При цьому у WSH як стандартні мови підтримуються VBScript та JScript.

Взагалі кажучи, принцип сценаріїв, що підтримуються WSH, полягає у використанні об’єктів ActiveX, тому спочатку ми коротко ознайомось із функціями технології ActiveX компанії Microsoft.

Можна нагадати, що Windows з самого початку розробила технологію зв’язування та вбудовування об’єктів (OLE) для забезпечення обміну даними між програмами. Спочатку технологію OLE використовували для створення складених документів, а потім для вирішення більш загального завдання – забезпечення застосунків власними функціями (послугами) один одного та правильним використанням цих функцій. Технологія, яка дозволяє одній програмі (клієнт автоматизації) викликати функції іншої програми (сервер автоматизації), називається автоматизацією OLE. OLE та OLE Automation засновані на базовій «компонентній» технології Component Object Model (COM), що розроблена Microsoft.

Як правило, компонентне програмне забезпечення – це метод розробки програмного забезпечення, який використовує технологію для створення програмних модулів, подібно до технології, що використовується для розробки апаратного забезпечення. Складні схеми складаються із стандартизованих мікросхем з чітко визначеними функціями документації. Розробники можуть ефективно використовувати такі мікросхеми, не враховуючи їх внутрішню структуру. У програмному компоненті, написаному будь-якою мовою програмування, деталі реалізації використовуваного алгоритму також приховані всередині компонента (об’єкта), а на поверхні є загальнодоступні інтерфейси, які можуть використовуватись іншими програмами, написаними тією самою чи іншою мовою.

На сьогодні термін OLE використовується лише з історичних причин. Натомість Microsoft використовує новий термін ActiveX з 1996 року, який спочатку означав компоненти (об'єкти) WWW (World Wide Web), створені на основі технології COM.

Технологія ActiveX вже давно є ключовою у продуктах Microsoft. Вона найбільш повно впроваджена у Microsoft Office, Internet Explorer та Internet Information Service (IIS). Ці продукти мають вбудований інтерпретатор мови сценаріїв для управління відповідними об'єктами автоматизації: VBScript (використовується в Microsoft Office, Internet Explorer, IIS) та JScript (використовується в Internet Explorer, IIS). Однак безпосередньо в операційних системах поза цих продуктів, сценарії, написані VBScript або JScript, не могли виконуватися.

Сервер сценаріїв WSH – це потужний інструмент, який забезпечує єдиний інтерфейс (об'єктну модель) для спеціальних мов (VBScript, JScript, PerlScript, REXX, TCL, Python тощо), щоб можна було використовувати будь-який зовнішній об'єкт ActiveX. За допомогою WSH сценарії можна виконувати безпосередньо в операційній системі Windows без вбудовування у HTML-сторінки.

WSH встановлює мінімальні вимоги до оперативної пам'яті і є дуже зручним інструментом для автоматизації щоденних завдань користувачів та адміністраторів Windows. Використовуючи сценарії WSH, ви можете безпосередньо використовувати файлову систему комп'ютера або керувати роботою інших програм (серверів автоматизації). Однак функціональність сценарію обмежується лише методами, доступними на доступному сервері автоматизації.

Ось найбільш очевидні завдання, для яких сценарії WSH чудово підходять для автоматизації:

1. Організація створення резервних копій файлів з локального комп'ютера на мережевий сервер, які можна відібрати за будь-якими умовами.
2. Швидка зміна конфігурації робочого столу Windows відповідно до виконуваного завдання.
3. Автоматичний запуск програми Microsoft Office, створення в ній складних документів та їх друк.
4. Контроль роботи програм, які не є серверами автоматизації, шляхом надсилання цим програмам натискання клавіш.
5. Підключення і відключення мережевих ресурсів (дисків і принтерів).
6. Створення складних сценаріїв реєстрації для користувачів.
7. Виконання завдань з управління локальною мережею (наприклад, додавання або видалення користувачів).

2.1. Створення і запуск найпростіших сценаріїв WSH

Найпростіший сценарій WSH, написаний на ***JScript*** або ***VBScript***, – це звичайний текстовий файл із розширенням ***js*** або ***vbs***, відповідно, і його можна створити в будь-якому текстовому редакторі, який може зберігати документи у текствому форматі.

Розмір сценарію може коливатися від одного до кількох тисяч рядків, а максимальний розмір обмежується лише максимальним розміром файлу у відповідній файловій системі.

Як перший приклад, давайте створимо сценарій JScript, який відображатиме діалогове вікно з написом «Привіт!» Для цього просто використовуйте, наприклад, стандартний блокнот Windows (notepad.exe), щоб створити файл ***example1.js***, який містить лише один рядок:

WScript.Echo("Привіт!");

Звичайно, той самий сценарій у VBScript має інший синтаксис, як показано нижче:

WScript.Echo "Привіт!"

Хоча для роботи цих двох сценаріїв достатньо одного рядка, рекомендується звикнути додавати інформацію про сценарій на початку файлу: назву мови, якою його написано, та короткий опис проведеної операції. У ***JScript*** ця інформація може бути оформленою у вигляді коментарів:

```
/***********************************************/
/* Ім'я: example1.js */
/* Мова: JScript */
/* Опис: Виведення на екран вітання */
/***********************************************/
```

У ***VBScript*** той самий вміст виглядає так:

```
'***********************************************
'Ім'я: example1.vbs
'Мова: VBScript
'Опис: Виведення на екран вітання
'***********************************************
```

Існує кілька способів запуску сценаріїв WSH.

Ви можете використовувати консольну версію WSH ***cscript.exe*** для запуску сценаріїв з командного рядка. Наприклад, щоб запустити сценарій, записаний у файлі C:\Script\First.js, потрібно відкрити командну оболонку і запустити у ній команду:

cscript C:\Script\First.js

У результаті цього сценарію в командному вікні відобразиться рядок «Привіт!».

Виконанням сценарію керують параметри командного рядка ***cscript.exe*** (табл. 2.1), що вмикає або вимикає різні параметри WSH (всі ці параметри починаються з символів *//*).

Табл. 2.1

Параметри командного рядка ***cscript.exe***

Параметр	Опис
//I	Вимкнути пакетний режим (за замовчуванням). Це покаже всі повідомлення про помилки в сценарії
//B	Увімкнути пакетний режим. У цьому випадку повідомлення про помилку в сценарії не відображатиметься на екрані
//T:nn	Встановити час очікування (у секундах), тобто сценарій буде працювати протягом *nn* секунд, після чого процес буде перерваний. За замовчуванням час виконання необмежений
//logo	Виводити (за замовчуванням) версію WSH та інформацію про розробника перед виконанням сценарію
//nologo	Заборонити виведення версії WSH та інформації про розробника
//H:CScipt або *//H:WScript*	Зробити *cscript*.exe або *wscript.exe* програмою, яка запускає сценарій за замовчуванням. Якщо ці характеристики не вказані, за замовчуванням використовується *wscript.exe*
//S	Зберігати налаштування командного рядка для поточного користувача
//?	Відображати вбудовані підказки для параметрів командного рядка
//E:engine	Виконувати сценарій, використовуючи модуль, заданий параметром *engine*
//D	Увімкнути налагоджувач
//X	Виконати програму у налагоджувачі
//Job:<JobID>	Запустити завдання з індексом *JobID* з багатозадачного файлу WSH

Наприклад, команда ***cscript //B my.js /a /b*** запустить сценарій ***my.js*** в пакетному режимі, де ***/a*** та ***/b*** – параметри цього сценарію, а ***//B*** – параметри програми cscript.exe.

Графічну версію WSH wscript.exe можна використовувати для виконання сценаріїв із командного рядка. Для прикладу в цьому випадку необхідно виконати команду:

wscript C:\Script\First.js

Тоді в результаті виконання сценарію на екрані з'явиться потрібне нам діалогове вікно. Тому, коли ви запускаєте сценарій у консольному режимі, виведення текстової інформації надходить у стандартний вихідний потік (на консоль), а коли ви запускаєте його графічно – у діалогове вікно.

Щоб запустити сценарій за допомогою пункту «Виконати» меню «Пуск», просто введіть повне ім'я сценарію в поле «Відкрити» та натисніть «ОК». У цьому випадку сценарій за замовчуванням буде виконаний за допомогою ***wscript.exe***, тобто вихідну інформацію буде виведено у графічному діалоговому вікні.

Найпростіший спосіб – запустити сценарій у Провіднику Windows або на робочому столі: просто двічі клацнути ім'я файлу сценарію або його піктограму (подібно до будь-якого іншого виконуваного файлу). У цьому випадку, як і у випадку використання меню «Пуск» для запуску, сценарій за замовчуванням буде виконано за допомогою ***wscript.exe***.

2.2. Мови VBScript і JScript для сценаріїв WSH

Як зазначалося вище, сценарії WSH можна писати будь-якою мовою, що підтримує технологію сценаріїв ActiveX. Однак для всіх таких мов (крім VBScript та JScript) у системі повинні бути встановлені інші модулі підтримки (бібліотеки). Ось чому мови VBScript або JScript найчастіше використовуються під час роботи з WSH для забезпечення того, щоб сценарії на цих мовах могли запускатися на будь-якому комп'ютері з операційною системою Windows.

Ці дві мови сильно відрізняються одна від одної з точки зору синтаксису та стилю програмування. ***JScript*** – це інтерпретована об'єктно-орієнтована мова сценаріїв, що була розроблена корпорацією Microsoft та спочатку використовувалась для створення динамічних HTML-сторінок. Зверніть увагу: хоча синтаксис JScript подібний до Java та C, вона не є усіченою версією будь-якої іншої мови програмування.

VBScript (Visual Basic Script Edition) – це полегшена версія Microsoft Visual Basic, тому ви будете добре знайомі з нею, якщо до цього використовували мови VBScript Visual Basic або VBA (Visual Basic for Applications).

Якою мовою користуватися, більше залежить від смаку та звичок. Якщо ви пишете мовою C або Java, швидше за все, станете використовувати ***JScript***. Якщо ви використовуєте Visual Basic або

VBA (Visual Basic для додатків) для програмування, то ви будете добре знайомі з мовою ***VBScript***.

Також зауважте, що у виборі мови для написання сценаріїв, які залучають зовнішні об’єкти (майже у всіх сценаріях Windows), під час використання ***VBScript*** ви можете додавати деякі інші параметри.

Перш за все, ***VBScript*** дозволяє переглядати елементи колекції безпосередньо в циклі ***For Each ... Next*** (такі колекції, як правило, властивості зовнішніх об’єктів). З цієї причини ви повинні використовувати допоміжний об’єкт ***Enumerator*** і цикл ***for*** з умовою завершення і оператором ітерації.

Розглянемо такий приклад. Припустимо, що змінна ***Folder*** є об’єктом, що відповідає кореневому каталогу диска ***C:***, тобто ***Folder*** – це колекція, що містить файлові об’єкти, розташовані в кореневому каталозі. Потім пошук усіх цих файлів у ***VBScript*** організовується таким чином:

```
'Створюємо колекцію Files всіх файлів в кореневому каталозі диска C:
Set Files = Folder.Files
'Перебираємо всі елементи колекції Files
For Each File In Files
  'Виділяємо ім'я файлу для поточного елемента File колекції
  s = s & File.Name & vbCrLf
Next
```

Подібний код у ***JScript*** виглядає так:

```
// Створюємо колекцію файлів
Files = new Enumerator(Folder.Files);
// Цикл по всіх файлів
for (; !Files.atEnd(); Files.moveNext())
  // Додаємо рядок з іменем файлу
  s + = Files.item().Name+"\n";
// Виводимо отримані рядки на екран
```

По-друге, ***VBScript*** дозволяє безпосередньо викликати методи об’єктів ***WMI***, тоді як у ***JScript*** ви повинні використовувати спеціальний об’єкт ***SWbemObject*** та об’єкти ***InParam*** (включають параметри методу виклику) й ***OutParam*** (параметри, що формуються після виконання методу).

Нарешті, більшість прикладів сценаріїв, які можна знайти в документах Microsoft або Інтернеті, написаних у ***VBScript***, можна використовувати під час розробки без змін і не витрачаючи час на переклад коду іншою мовою. Детальне обговорення синтаксису ***VBScript*** та ***JScript*** виходить за рамки цього посібника. У наведених нижче прикладах сценаріїв ми будемо використовувати ***JScript***.

2.2.1. Основні відомості про мову Jscript

JScript – це об’єктно-орієнтована інтерпретована мова сценаріїв Microsoft, що спочатку розроблялась для створення динамічних HTML-сторінок. Зверніть увагу: хоча синтаксис ***JScript*** подібний до ***Java*** та ***C***, він не є усіченою версією будь-якої іншої мови програмування. Далі ми коротко представимо функції ***JScript***, які можуть знадобитися під час написання сценаріїв, які працюють з WSH.

Як і у будь-якій іншій мові програмування, у ***JScript*** ви можете використовувати змінні, називаючи їх на ім’я. У цьому випадку змінні можуть бути глобальними (доступними з будь-якого місця сценарію) або локальними (область дії обмежена функцією, в якій вони визначені).

Хорошим тоном є попереднє оголошення змінних, що використовуються з ключовим словом ***var***, хоча це лише обов’язкова умова для локальних змінних, визначених у функції. Приклад оголошення змінної такий:

var MyVariable;

У оголошенні не вказано явно тип змінної (як, наприклад, це робиться в мові ***C*** або ***Pascal***). Тільки коли значення записується у змінну, їм присвоюється певний тип. Мова ***JScript*** чутлива до регістру, що означає, що імена ***MyVariable*** та ***myvariable*** представляють різні змінні. Крім того, під час вибору імен змінних слід дотримуватися таких правил:

– ім’я змінної повинно починатися з літери або літери "_", "***$***" і може складатися лише з літер, цифр і символів "_", "***$***";

– імена змінних не повинні відповідати зарезервованим ключовим словам ***JScript***.

JScript підтримує шість типів даних, найважливішими з яких є числа, рядки, об’єкти та логічні дані. Останні два типи – ***null*** (порожній тип) та ***undefined*** (невизначений тип).

Як і в інших алгоритмічних мовах, функції в ***JScript*** дозволяють поєднувати кілька операцій з одним іменем. За необхідності функцію можна викликати з будь-якого місця сценарію. Сценарій ***JScript*** підтримує два типи функцій: вбудовані функції та функції користувача, написані нами самими. Визначені користувачем функції можуть бути в будь-якому місці сценарію та мати будь-яку кількість параметрів (аргументів). Загальне визначення цієї функції є таким:

function Ім’я_функції([параметр1] [,параметр2] [...,параметри])
{
…

тіло функції
…
[return значення;]
}

Ключове слово ***return*** дозволяє функціям повертати значення будь-якого допустимого типу. Наприклад, якщо обидва параметри наведеної нижче функції ***MyFunction*** менше 10, вона повертає ***true***:

```
function MyFunction(x,y)
{
    if ((x<10) && (y<10))
        return true
    else
        return false;
}
```

Вбудовані об’єкти (класи)

Як згадувалося вище, ***JScript*** є об’єктно-орієнтованою мовою, тому ви можете використовувати відповідні вбудовані об’єкти для обробки рядків, дат і часу, а також таких структур, як масиви та колекції. Крім того, для виконання математичних розрахунків можна використовувати спеціальні вбудовані об’єкти. У табл. 2.2 представлені деякі об’єкти, які можуть бути корисними під час створення сценаріїв за допомогою WSH.

Табл. 2.2

Деякі вбудовані об’єкти **Jscript**

Об’єкт	Опис
Array	Створення і робота з масивами даних довільного типу
Date	Робота з даними, що містять дату або час
Enumerator	Робота з колекціями даних довільного типу
Math	Виконання математичних обчислень
String	Робота з текстовими рядками

Для того, щоб використовувати вбудований об’єкт у сценарії, ви повинні створити змінну, яка дозволить вам отримати доступ до властивостей та методів об’єкта. Для створення більшості змінних цього типу потрібно використовувати новий оператор і спеціальну функцію-конструктор потрібного об’єкта. Ім’я конструктора завжди збігається з іменем відповідного вбудованого об’єкта. Це приклад створення об’єктів ***Date*** та ***Array***:

```
var d;
d = new Date();
var a;
a = new Array(10);
```

Зверніть увагу, що ви можете створити об'єкт String, просто заключивши значення рядка в лапки:

```
var s;
s = "Привіт!";
```

Об'єкт Array

Новий об'єкт вбудованого класу ***Array*** можна створити за допомогою оператора ***new*** такими способами:

- ***new Array()*** – створити масив нульової довжини;
- ***new Array(N)*** – створити масив довжини ***N***;
- ***new Array(a0, a1, ..., aN)*** – створити масив довжини ***N+1*** з елементами ***a0, ai, ..., aN***.

Наприклад:

```
var A1, A2, A3;
A1 = new Array();
A2= new Array(3);
A3 = new Array(0,"Рядок",2.5};
```

Нумерація елементів у масиві завжди починається з нуля. Після створення та ініціалізації масиву ви можете використовувати звичайний оператор індексації ***[]*** для доступу до його елементів, наприклад:

```
A3[1] = A3[0] + A3[2];
```

Довжину масиву, тобто кількість елементів, з яких він складається, можна знайти, використовуючи властивість ***length*** об'єкта ***Array***. Щоб динамічно змінювати довжину масиву (зменшувати або збільшувати), просто впишіть відповідне значення у властивість ***length***:

```
var A;
A = new Array(1,2,3,4,5); //Довжина масиву А дорівнює 5
A.length = 3; //Тепер довжина масиву А дорівнює 3
```

Об'єкт Date

Один із трьох конструкторів, що представлені нижче, використовується для створення нового об'єкта вбудованого класу ***Date***.

Перший тип конструктора дозволяє створити об'єкт, що зберігає інформацію про поточну дату та час:

var d;
d = new Date();

Тут час встановлюється відповідно до Грінвічського стандарту, використовуючи Coordinated Universal Time (UTC).

Другий тип конструктора має один параметр:

var d;
d = new Date(nMilliseconds);

Параметр ***nMilliseconds*** задає дату в мілісекундах, рахуючи від 1 січня 1970 року.

Третій тип конструктора призначений для використання окремого завдання компонента дати, і його вигляд такий:

var d;
d = New Date (year, month, date [, hours [, mm [, sec [, ms]]]]);

Об'єкт Enumerator

Ви можете використовувати об'єкт Enumerator для доступу до будь-якого елемента колекції (у ***VBScript*** для цього використовується цикл ***For .. Each***). Колекція в ***JScript*** – це група елементів. Відмінність від масиву полягає в тому, що ви можете використовувати індекс для безпосереднього доступу до елементів колекції – ви можете переміщати лише вказівник поточного елемента на перший або наступний відносно поточного елементу.

Щоб створити новий об'єкт вбудованого класу ***Enumerator***, використовуйте такий конструктор:

var e;
e = new Enumerator(collection);

Тут параметр ***collection*** вказує на колекцію для доступу до елементів, що створені як елементи класу ***Enumerator***. Сама колекція, як правило, є властивістю інших об'єктів. Методи об'єкта ***Enumerator*** перераховані в табл. 2.3. Цей об'єкт не має властивостей.

Табл. 2.3

Методи об'єкта ***Enumerator***

Метод	Опис
atEnd()	Якщо вказівник на поточний елемент знаходиться на елементі після останнього екземпляра колекції, або колекція порожня, або поточний елемент не визначений, повертає *true*. В іншому випадку повертає *false*.
item()	Повертає значення поточного елемента колекції. Якщо колекція порожня або поточний елемент не визначений, повертає значення *undefined*
moveFirst()	Переміщує вказівник на перший елемент у колекції. Якщо в колекції немає елементів, поточний елемент приймає значення *undefined*
moveNext()	Переміщує вказівник на наступний елемент у колекції. Якщо вказівник знаходиться на останньому елементі колекції перед використанням цього методу, або колекція порожня, поточний елемент приймає значення *undefined*

Розглянемо такий приклад використання класу ***Enumerator***:

```
var fso, s, n, e, x;
//Створення об'єкта FileSystemObject
fso = WScript.CreateObject("Scripting.FileSystemObject");
//Створення об'єкта Enumerator для доступу до колекції fso.Drives
e = new Enumerator(fso.Drives);
s = "";
// Цикл для перегляду всіх елементів колекції
for (; !e.atEnd(); e.moveNext()) {
    //Витягнення елементів колекції
    x = e.item() ;
    s = s + x.DriveLetter;
    s += " - ";
    if (x.DriveType == 3)
        n = x.ShareName;
    else
        if (x.IsReady)
            n = x.VolumeName;
        else
            n = "[Пристрій не готовий]";
    s += n + "\n";
}
```

Тут використовується метод ***CreateObject*** об'єкта ***WScript*** для створення екземпляра об'єкта ***FileSystemObject***, який використовується для доступу до файлової системи комп'ютера. Властивість ***Drives*** цього об'єкта – це колекція, що містить інформацію про всі доступні пристрої (диски).

У циклі ми формуємо рядок ***s***, де будуть записані літери всіх дисків комп'ютера та імена цих дисків (якщо це локальний диск, то це мітка диска; якщо це мережа, то це ім'я диска в UNC-форматі).

2.3. Власна об'єктна модель WSH

Далі опишемо власну об'єктну модель ***Windows Script Host***. Ви можете використовувати внутрішні об'єкти WSH у сценарії для виконання таких основних завдань:

– відображення інформації у стандартному вихідному потоці (на екрані) або у діалоговому вікні Windows;

– зчитування даних зі стандартного вхідного потоку (тобто введення даних з клавіатури) або використання інших команд для відображення інформації;

– використання властивостей та методів зовнішніх об'єктів та обробка подій, породжених цими об'єктами;

– запуск нового незалежного процесу або активація вже існуючого процесу;

– запустіть дочірнього процесу для контролю його стану і отримання доступу до його стандартних потоків введення та виведення;

– робота з локальною мережею: визначення імені зареєстрованого користувача, підключення до мережевих дисків та принтерів;
– перегляд змінних середовища;
– доступ до спеціальних папок Windows;
– створення ярликів Windows;
– використання системного реєстру.

WSH містить такі об’єкти, що перераховані у табл. 2.4.

Табл. 2.4

Об’єкти WSH

Об’єкт	**Призначення**
WScript	Основний об’єкт WSH, який використовується для створення або зв’язку з іншими об’єктами, містить інформацію про сервер сценаріїв, а також дозволяє вводити дані з клавіатури та відображати інформацію на екрані або у вікні Windows
WshArguments	Надає доступ до всіх параметрів командного рядка запущених сценаріїв або ярликів Windows
WshNamed	Надає доступ до іменних параметрів командного рядка запущеного сценарію
WshUnnamed	Забезпечує доступ до безіменних параметрів командного рядка запущеного сценарію
WshShell	Дозволяє запускати незалежні процеси, створювати ярлики, використовувати змінні середовища, реєстр та спеціальні папки Windows
WshSpecialFolders	Забезпечує доступ до спеціальних папок Windows
WshShortcut	Дозволяє використовувати ярлики Windows
WshUrlShortcut	Дозволяє використовувати ярлики мережевих ресурсів
WshEnvironment	Призначений для перегляду, модифікації та видалення змінних середовища
WshNetwork	Використовується для роботи з локальною мережею: містить інформацію про мережу локального комп’ютера, дозволяє підключати мережеві диски та принтери
WshScriptExec	Дозволяє запускати консольні програми як дочірні процеси, забезпечуючи контроль за станом цих програм та доступ до їх стандартних потоків введення та виведення
WshController	Дозволяє запускати сценарії на віддалених комп’ютерах
WshRemote	Дозволяє керувати сценаріями, що працюють на віддалених комп’ютерах
WshRemoteError	Використовується для отримання інформації про помилки, спричинені запуском сценарію на віддаленому комп’ютері

Крім того, існує ***FileSystemObject***, який забезпечує доступ до файлової системи комп'ютера (цей об'єкт буде детально розглянуто далі).

2.3.1. Об'єкт WScript

Властивості об'єкта ***WScript*** дозволяють отримати повний шлях до використовуваного сервера сценаріїв (***wscript.***exe або ***cscript.exe***), параметри командного рядка для запуску сценарію та режим його роботи (інтерактивний або пакетний). Крім того, ви можете використовувати властивості об'єкта ***WScript*** для виведення інформації у стандартний вихідний потік та зчитування даних зі стандартного вхідного потоку. ***WScript*** також забезпечує спосіб роботи всередині сценарію з об'єктами автоматизації та відображення інформації на екрані (у текстовому режимі) або у вікні Windows.

Зверніть увагу, що у сценаріях ***WSH*** об'єкт ***WScript*** можна використовувати негайно без попереднього опису чи створення, оскільки його екземпляр автоматично створюється сервером сценаріїв. Для роботи всіх інших об'єктів потрібно використовувати метод ***CreateObject***. Властивості об'єкта ***WScript*** показані в табл. 2.5.

Табл. 2.5

Властивості об'єкта ***WScript***

Властивість	**Опис**
Arguments	Містить вказівник на колекцію *WshArguments*, яка містить параметри командного рядка для виконуваних сценаріїв
FullName	Містить повний шлях до виконуваного файлу сервера сценаріїв (в Windows зазвичай це *C:\WINDOWS\SYSTEM32\CSCRIPT.EXE* або *C:\WINDOWS\SYSTEM32\WSCRIPT.EXE*)
Name	Містить назву об'єкта *WScript* (Windows Script Host)
Path	Містить шлях до каталогу, що містить *cscript.exe* або *wscript.exe* (в Windows зазвичай це *C:\WINDOWS\SYSTEM32*)
ScriptFullName	Містить повний шлях до запущеного сценарію
ScriptName	Містить ім'я запущеного сценарію
StdErr	Дозволяє запущеному сценарію записувати повідомлення в стандартний потік для помилок
StdIn	Дозволяє запущеному сценарію читати інформацію зі стандартного вхідного потоку
StdOut	Дозволяє запущеному сценарію записувати інформацію в стандартний вихідний потік
Version	Містить версію WSH

Опишемо більш детально деякі властивості об'єкта ***WScript***.

Властивість Arguments

У наступному прикладі на екран виводяться всі параметри командного рядка, з якими був запущений сценарій.

```
var e, s, objArgs, Arg
objArgs = WScript.Arguments; //Створюємо об'єкт WshArguments
e = new Enumerator(objArgs);
s="";
for (var i=0; i<objArgs.length; i++) {
    s+=objArgs(i)+" "; //Формуємо рядок зі значеннями аргументів
}

WScript.Echo(s); //Виводимо сформований рядок
```

Властивості StdErr, StdIn, StdOut

Властивості ***StdIn***, ***StdOut*** та ***StdErr*** можна використовувати для доступу до стандартних потоків введення та виведення лише тоді, коли сценарій запускається в консольному режимі за допомогою ***cscript.exe***. Якщо сценарій запущений за допомогою ***wscript.exe***, зі спробою отримати доступ до цих властивостей ви побачите помилку ***"Invalid Handle"***.

Ви можете використовувати методи ***Write***, ***WriteLine***, ***WriteBlankLines*** для використання потоків ***StdOut*** і ***StdErr***, а методи ***Read***, ***ReadLine***, ***ReadAll***, ***Skip***, ***SkipLine*** – для використання потоків ***StdIn***. Ці методи коротко описані в табл. 2.6.

Табл. 2.6

Методи для роботи зі стандартними потоками

Метод	Опис
Read(n)	Зчитує кількість символів, вказану параметром *n*, з потоку *StdIn* і повертає отриманий рядок
ReadAll()	Зчитує символи з потоку StdIn, поки не зустрінеться кінцевий символ файлу ASCII 26 (<Ctrl>+<Z>), і повертає отриманий рядок
ReadLine()	Повертає рядок, зчитаний з потоку *StdIn*
Skip(n)	Пропускає при читанні з потоку *StdIn* задане параметром n число символів
SkipLine()	Пропускає цілий рядок при читанні з потоку *StdIn*
Write(string)	Записує в потік *StdOut* або *StdErr* рядок *string* (без символу кінця рядка)
WriteBlankLines(n)	Записує в потік *StdOut* або *StdErr* заданий параметром n число порожніх рядків
WriteLine(string)	Записує в потік *StdOut* або *StdErr* рядок string (разом з символом кінця рядка)

Нагадаємо, що операційна система Windows підтримує механізм конвеєризації (символ "/" в командному рядку). Цей механізм може

передавати дані з однієї програми в іншу. Отже, використовуючи стандартні вхідні та вихідні потоки, ви можете обробити вихідний рядок іншої програми в сценарії або перенаправити дані, що виводяться сценарієм, на вхід програм-фільтрів (***FIND*** або ***SORT***). Наприклад, наступна команда сортує вихідні рядки сценарію ***example.js*** і виводить їх у файл ***sort.txt***:

cscript //Nologo example.js | sort > sort.txt

Тут потрібна опція //Nologo, щоб файл sort.txt не містив рядків, що містять інформацію про розробника та номер версії WSH.

Крім того, використовуючи метод, що працює з вхідним потоком StdIn, ви можете організовувати діалог з користувачами, тобто створювати інтерактивні сценарії. Відповідний приклад наведено нижче.

```
//Виводимо рядок на екран
WScript.StdOut.Write("Введіть рядок:");
//Прочитуємо рядок
s = WScript.StdIn.ReadLine();
//Виводимо рядок на екран
WScript.StdOut.WriteLine("Ви ввели число" + s);
```

Методи об'єкта WScript

Об'єкт ***WScript*** має кілька методів, які описані в табл. 2.7.

Табл. 2.7

Методи об'єкта ***WScript***

Метод	Опис
CreateObject(strProgID [,strPrefix])	Створює об'єкт, зазначений параметром *strProgID*
ConnectObject(strObject, strPrefix)	Встановлює зв'язок з об'єктом *strObject*, дозволяючи писати обробники функцій для його подій (імена цих функцій повинні починатися з префікса *strPrefix*)
DisconnectObject(obj)	Розриває зв'язок з об'єктом *obj*, який раніше був підключений у сценарії
Echo([Arg1] [, Arg2] [...])	Відображує текстову інформацію в консолі або діалоговому вікні
GetObject(strPathname [, strProgID], [strPrefix])	Активує об'єкт автоматизації, визначений вказаним файлом (параметр *strPathName*), або об'єкт, зазначений параметром *strProgID*
Exit([intErrorCode])	Використовує вказаний вихідний код *intErrorCode*, щоб перервати виконання сценарію. Якщо параметр *intErrorCode* не вказаний, об'єкт *WScript* встановлює вихідний код на нуль

Закінчення табл.

Sleep(IntTime)	Зупиняє виконання сценарію (переводить його в неактивному стані) на заданому параметрі *IntTime* число мілісекунд

Наведемо додаткові пояснення і приклади використання для методів, наведених в таблиці.

Метод CreateObject

Рядковий параметр strProgID, зазначений у методі CreateObject, називається програмним ідентифікатором об’єкта (Programmic ProgID).

Якщо вказано необов’язковий параметр ***strPrefix***, після створення об’єкта в сценарії ви можете обробляти події, що відбуваються в об’єкті (звичайно якщо об’єкт надає інтерфейс для зв’язку з цими подіями). Коли об’єкт повідомляє про подію, сервер сценаріїв викликає функцію, ім’я якої складається з префікса ***strPrefix*** та імені події. Наприклад, якщо ***"MYOBJ_"*** вказано як ***strPrefix***, і об’єкт повідомляє про появу події "OnBegin", буде запущено функцію ***"MYOBJ_OnBegin"***, яка повинна бути описана в сценарії. У наступному прикладі використовується метод ***CreateObject*** для створення об’єкта ***WshNetwork***:

```
var WshNetwork = WScript.CreateObject("WScript.Network");
```

Зверніть увагу, що об’єкти автоматизації в сценарії можна створювати без допомоги WSH. ***JScript*** використовує для цього ***ActiveXObject***, наприклад:

```
var WshNetwork = new ActiveXObject("WScript.Network");
```

У ***VBScript*** ви можете використовувати спеціальну функцію ***CreateObject*** для створення об’єктів, наприклад:

```
Set WshNetwork = CreateObject("WScript.Network");
```

Однак лише тоді, коли використовується метод ***WScript. CreateObject***, обробка подій створеного об’єкта може бути організована в сценарії.

Метод ConnectObject

Об’єкт, підключений за допомогою методу ***ConnectObject***, повинен надавати інтерфейс для своїх подій.

У наступному прикладі у змінній ***MyObject*** створюється абстрактний об’єкт ***"SomeObject"***, а потім зі сценарію викликається метод ***SomeMethod*** цього об’єкта. Після цього встановлюється зв’язок зі змінною ***MyObject*** та задається префікс ***"MyEvent"*** для обробки подій цього об’єкта. Якщо в об’єкті відбувається подія з назвою "подія", буде викликана функція ***MyEvent_Event***. Метод ***Disconnect Object*** об’єкта ***WScript*** відключає об’єкт ***MyObject***.

```
var MyObject = WScript.CreateObject("Деякий об'єкт");
MyObject.someMethod();
WScript.ConnectObject(MyObject, "MyEvent");

function MyEvent_Event(strName)
{
  WScript.Echo(strName);
}

WScript.DisconnectObject(MyObject);
```

Метод Echo

Параметри ***arg1, arg2, ...*** методу ***Echo*** визначають вихідні параметри. Якщо сценарій запускається за допомогою ***wscript.exe***, метод ***Echo*** направлятиме вихід у діалогове вікно; якщо ви використовуєте ***cscript.exe*** для виконання сценарію, результат буде спрямований на екран (консоль). Кожен параметр на виході буде розділений пробілом. Якщо ви використовуєте ***cscript.exe***, виведення усіх параметрів закінчуватиметься символом нового рядка. Якщо в методі ***Echo*** не вказано жодного параметра, буде видано порожній рядок.

Наприклад, після запуску сценарію ***EchoExample.js*** через ***cscript.exe*** на екрані відображатиметься порожній рядок, три цифри та рядок тексту.

```
WScript.Echo(); //Виводимо порожній рядок
WScript.Echo(1,2,3); //Виводимо числа
WScript.Echo("Привіт!"); //Виводимо рядок
```

Метод Sleep

У наступному прикладі сценарій неактивний протягом 5 секунд:

```
WScript.Echo("Сценарій запущений, відпочиває ...");
WScript.Sleep(5000);
WScript.Echo("Виконання сценарію завершено");
```

Наприклад, при використанні методу ***WshShell.SendKeys*** для імітації натискань клавіш в активному вікні метод ***Sleep*** повинен використовуватися для асинхронних операцій сценаріїв та будь-яких інших завдань.

2.3.2. Об'єкт WshShell

За допомогою об'єкта ***WshShell*** можна запускати новий процес, створювати ярлики, працювати з системним реєстром, отримувати доступ до змінних середовища і спеціальним папок Windows. Створюється цей об'єкт таким чином:

var WshShell = WScript.CreateObject("WScript.Shell");

Об'єкт ***WshShell*** має три властивості, які наведені у табл. 2.8.

Табл. 2.8

Властивості об'єкта ***WshShell***

Властивість	Опис
CurrentDirectory	Тут зберігається повний шлях до поточного каталогу (до каталогу, з якого був запущений сценарій)
Environment	Містить об'єкт ***WshEnvironment***, який забезпечує доступ до змінних середовища операційної системи для Windows NT/2000 /XP або до змінних середовища поточного командного вікна для Windows 9x
SpecialFolders	Містить об'єкт ***WshSpecialFolders*** для доступу до спеціальних папках Windows (робочий стіл, меню Пуск (Start) та ін.)

Наведемо тепер методи, наявні в об'єкта ***WshShell***.

Табл. 2.9

Методи об'єкта ***WshShell***

Метод	Опис
AppActivate(title)	Активізує заданий параметром title вікно програми. Рядок title задає назву вікна (наприклад, "calc" або "notepad") або ідентифікатор процесу (Process ID, PID)
CreateShortcut(strPathname)	Створює об'єкт ***WshShortcut*** для зв'язку з ярликом Windows (розширення lnk) або об'єкт ***WshUrlShortcut*** для зв'язку з мережевим ярликом (розширення url). Параметр ***strPathname*** задає повний шлях до створюваного або змінюваного ярлика
Environment(strType)	Повертає об'єкт ***WshEnvironment***, в якому знаходяться середовища заданого виду

Закінчення табл.

Exec(strCommand)	Створює новий дочірній процес, який запускає консольний додаток, заданий параметром ***strCommand***. У результаті повертається об'єкт ***WshScriptExec***, що дозволяє контролювати хід виконання запущеного додатку і забезпечує доступ до водних потоків ***StdIn***, ***StdOut*** і ***StdErr*** цього додатка
ExpandEnvironmentStrings(strString)	Повертає значення змінної середовища поточного командного вікна, заданим рядком ***strString*** (ім'я змінної повинно бути оточене знаками "%")
LogEvent(intType, strMessage[, strTarget])	Протоколює події в журналі Windows або в файлі WSH.log. Цілочисельний параметр ***intType*** визначає тип повідомлення, рядок ***strMessage*** – текст повідомлення. Параметр ***strTarget*** він визначає назву системи, в якій записуються події (за замовчуванням це локальна система). Метод ***LogEvent*** повертає ***true***, якщо подія записано успішно і ***false*** в іншому випадку
Popup(strText, [nSecToWait], [strTitle], [nType])	Виводить на екран інформаційне вікно з повідомленням, заданим параметром ***strText***. Параметр ***nSecToWait*** задає кількість секунд, після закінчення яких вікно буде автоматично закрито, параметр ***strTitle*** визначає заголовок вікна, параметр ***nType*** вказує тип кнопок і значка для вікна
RegDelete(strName)	Видаляє з системного реєстру заданий параметр або розділ цілком
RegRead(strName)	Повертає значення параметра реєстру або значення за замовчуванням для розділу реєстру
RegWrite(strName, anyValue [, strType])	Записує до реєстру значення заданого параметра або значення за замовчуванням для розділу
Run(strCommand, [intWindowStyle], [bWaitOnReturn])	Створює новий незалежний процес, який запускає додаток, заданий параметром strCommand
SendKeys(string)	Посилає одне або кілька натискань клавіш в активне вікно (ефект той же, якщо б ви натискали ці клавіші на клавіатурі)
SpecialFolders(strSpecFolder)	Повертає рядок, що містить шлях до спеціальної папки Windows, заданої параметром ***strSpecFolder***

Розглянемо більш докладно деякі методи, наведені в таблиці.

Метод AppActivate

Метод AppActivate активує вже запущений вказаний додаток (встановлює на нього фокус), але не виробляє ніяких дій зі зміни розмірів його вікна. Для того, щоб спочатку запустити потрібну програму і визначити вид його вікна, треба використовувати метод ***Run*** об'єкта ***WshShell***. Для того, щоб визначити, який самий додаток необхідно активізувати, рядок title порівнюється по черзі з назвами вікон всіх запущених додатків. Якщо не знайдено жодного точного збігу, буде проводитись пошук того додатка, назва вікна якого починається з рядка ***title***. Якщо і в цьому випадку не буде знайдено жодного підходящого додатка, то буде вестися пошук додатка, заголовок якого закінчується на цей рядок. Якщо буде буде знайдено декілька відповідних вікон, то відбудеться активізація одного з них (вікно обирається довільно).

Метод CreateShortcut

Цей метод дозволяє створити новий або відкрити вже існуючий ярлик для зміни його властивостей.

Наведемо приклад сценарію, в якому створюються два ярлика – на сам виконуваний сценарій (об'єкт ***oShellLink***) і на мережевий ресурс (***oUrlLink***).

```
var WshShell, oShellLink, oUrlLink;
//Створюємо об'єкт WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Створюємо ярлик на файл
oShellLink = WshShell.CreateShortcut("Current Script.lnk");
//Встановлюємо шлях до файлу
oShellLink.TargetPath = WScript.ScriptFullName;
//Зберігаємо ярлик
oShellLink.Save();

//Створюємо ярлик на мережевий ресурс
oUrlLink = WshShell.CreateShortcut("Microsoft Web Site.URL");
//Встановлюємо URL
oUrlLink.TargetPath = "http://www.microsoft.com";
//Зберігаємо ярлик
oUrlLink.Save();
```

Метод Popup

Якщо в методі не заданий параметр ***strTitle***, то заголовком вікна буде "Windows Script Host".

Параметр ***nType*** може приймати ті ж значення, що і в функції ***MessageBox*** з Microsoft Win32 API. У табл. 2.10 описані деякі можливі значення параметра ***nType*** і їх зміст.

Табл. 2.10

Типи кнопок та іконок для методу ***Popup***

Значення nType	Опис
0	Виводиться кнопка **Ok**
1	Виводяться кнопки **Ok** і **Cancel**
2	Виводяться кнопки **Abort**, **Retry** і **Ignore**
3	Виводяться кнопки **Yes**, **No** і **Cancel**
4	Виводяться кнопки **Yes** і **No**
5	Виводяться кнопки **Retry** і **Cancel**
16	Виводиться іконка **Stop Mark**
32	Виводиться іконка **Question Mark**
48	Виводиться іконка **Exclamation Mark**
64	Виводиться іконка **Information Mark**

Природно, що в методі ***Popup*** можна комбінувати значення параметра, наведені в табл. 6.3.

Метод ***Popup*** повертає ціле значення, за допомогою якого можна дізнатись, яка саме кнопка була натиснута для виходу (табл. 2.11).

Табл. 2.11

Значення, що повертаються методом ***Popup***

Значення	Опис
-1	Користувач не натиснув на жодну з кнопок протягом часу, заданого параметром ***nSecToWait***
1	Натиснута кнопка **Ok**
2	Натиснута кнопка **Cancel**
3	Натиснута кнопка **Abort**
4	Натиснута кнопка **Retry**
5	Натиснута кнопка **Ignore**
6	Натиснута кнопка **Yes**
7	Натиснута кнопка **No**

Метод Run

Параметр ***intWindowStyle*** встановлює вид вікна для додатку, що запускається (табл. 2.12).

Табл. 2.12

Типи вікна (***intWindowStyle***)

Параметр	Опис
0	Приховує поточне вікно і активує інше вікно
1	Активує і відображує вікно. Якщо вікно було мінімізовано або максимізовано, система відновить його початкове розташування і розмір. Цей прапорець повинен вказуватися сценарієм під час першого відображення вікна
2	Активує вікно і відображає його в мінімізованому (згорнутому) вигляді
3	Активує вікно і відображає його в максимізованому (розгорнутому) вигляді
4	Показує вікно в тому вигляді, в якому воно знаходилося останній раз. Активне вікно при цьому залишається активним
5	Активує вікно і відображує його в поточному стані
6	Мінімізує задане вікно і активує наступне (в Z-порядку) вікно
7	Показує вікно в згорнутому вигляді. Активне вікно при цьому залишається активним
8	Показує вікно в його поточному стані. Активне вікно при цьому залишається активним
9	Активує і відображає вікно. Якщо вікно було мінімізовано або максимізовано, система відновить його початкове розташування і розмір. Цей прапорець повинен вказуватися сценарієм, якщо виводиться відновлення згорнутого вікна
10	Встановлює режим відображення, що спирається на режим програми, яка запускає додаток

Необов’язковий параметр ***bWaitOnReturn*** є логічною змінною, дає вказівку очікувати завершення запущеного процесу. Якщо цей параметр не зазначений або встановлений у ***false***, то після запуску з сценарію нового процесу управління відразу ж повертається назад у сценарій (не чекаючи завершення запущеного процесу). Якщо ж ***bWaitOnReturn*** встановлений у ***true***, то сценарій відновить роботу тільки після завершення викликаного процесу.

При цьому якщо параметр ***bWaitOnReturn*** дорівнює ***true***, то метод ***Run*** повертає код виходу викликаного додатка. Якщо ж ***bWaitOnReturn*** дорівнює ***false*** або не заданий, то метод ***Run*** завжди повертає нуль.

У наступному прикладі ми запускаємо Microsoft Notepad і відкриваємо в ньому файл з виконуваним сценарієм:

```
var WshShell = WScript.CreateObject("WScript.Shell");
WshShell.Run("%windir%\\notepad " + WScript.ScriptFullName);
```

Наступний сценарій друкує код виходу викликаної програми:

```
var WshShell = WScript.CreateObject("WScript.Shell");
Return = WshShell.Run("notepad " + WScript.ScriptFullName, 1, true);
WScript.Echo("Код повернення:", Return);
```

Метод Send Keys

Кожна клавіша задається одним або декількома символами. Наприклад, для того, щоб задати натискання один за одним букв ***А***, ***Б*** і ***В***, потрібно вказати як параметр для ***SendKeys*** рядок "***АБВ***": ***string = "АБВ"***.

Кілька символів мають в методі ***SendKeys*** спеціальне значення: +, ^, ***%***, ~, *(*, *)*. Для того, щоб задати один з цих символів, їх потрібно укласти у фігурні дужки (*{}*). Наприклад, для завдання знака плюс використовується ***{+}***.

Квадратні дужки (*[]*) хоча і не мають в методі ***SendKeys*** спеціального сенсу, їх також потрібно укладати у фігурні дужки. Крім цього, для завдання самих фігурних дужок слід використовувати такі конструкції: ***{{}*** (ліва дужка) і ***{}}*** (права дужка).

Для завдання символів, що не відображаються, таких як ***<Enter>*** або ***<Tab>***, і спеціальних клавіш в методі ***SendKeys*** використовуються коди, представлені в табл. 2.13.

Табл. 2.13

Коди спеціальних клавіш для ***SendKeys***

Клавіша	Код	Клавіша	Код
<Backspace>	{BACKSPACE}, {BS} або {BKSP}	<Стрілка вниз>	{DOWN}
<Break>	{BREAK}	<Стрілка вправо>	{RIGHT}
<Caps Lock>	{CAPSLOCK}	<F1>	{F1}
<Del> або <Delete>	{DELETE} або {DEL}	<F2>	{F2}
<End>	{END}	<F3>	{F3}
<Enter>	{ENTER} або ~	<F4>	{F4}
<Esc>	{ESC}	<F5>	{F5}
<Help>	{HELP}	<F6>	{F6}
<Home>	{HOME}	<F7>	{F7}
<Ins> або<Insert>	{INSERT} або {INS}	<F8>	{F8}
<Num Lock>	{NUMLOCK}	<F9>	{F9}
<Page Down>	{PGDN}	<F10>	{F10}
<Page Up>	{PGUP}	<F11>	{F11}
<Print Screen>	{PRTSC}	<F12>	{F12}
<Scroll Lock>	{SCROLLLOCK}	<F13>	{F13}
<Tab>	{TAB}	<F14>	{F14}
<Стрілка вгору>	{UP}	<F15>	{F15}
<Стрілка вліво>	{LEFT}	<F16>	{F16}

Для завдання комбінацій клавіш з <***Shift***>, <***Ctrl***> або <***Alt***> перед відповідною клавішею потрібно поставити один або кілька кодів з табл. 2.14.

Табл. 2.14

Коди клавіш для <***Shift***>, <***Ctrl***> та <***Alt***>

Клавіша	**Код**
<Shift>	+
<Ctrl>	^
<Alt>	%

Для того, щоб задати комбінацію клавіш, яку потрібно набирати, утримуючи комбінацію клавіш <***Shift***>, <***Ctrl***> або <***Alt***>, необхідно укласти коди цих клавіш в дужки. Наприклад, якщо потрібно зімітувати натиснення клавіш <***G***> і <***S***>, утримуючи клавішу <***Shift***>, слід використовувати послідовність ***+(GS)***. Для того ж, щоб задати одночасне натиснення клавіш <***Shift***> і <***G***>, а потім <***S***> (вже без <***Shift***>), використовується ***+GS***.

У методі ***SendKeys*** можна задати кілька натискань поспіль однієї і тієї ж клавіші. Для цього необхідно у фігурних дужках вказати код потрібної клавіші, а через пробіл – число натиснень. Наприклад, ***{LEFT 42}*** означає натискання клавіші **<Стрілка вліво>** 42 рази поспіль; ***{H 10}*** означає натискання клавіші **<h>** 10 разів поспіль.

2.3.3. Об’єкт WshShortcut

За допомогою об’єкта ***WshShortcut*** можна створити новий ярлик Windows або змінити властивості вже існуючого ярлика. Цей об’єкт можна створити тільки за допомогою методу ***CreateShortcut*** об’єкта WshShell. У наступному прикладі зі сценарію створюється ярлик на цей сценарій (ярлик буде знаходитися в поточному каталозі):

```
var WshShell = WScript.CreateObject("WScript.Shell");
var oShellLink = WshShell.CreateShortcut("Current Script.lnk");
oShellLink.TargetPath = WScript.ScriptFullName;
oShellLink.Save();
```

Властивості об’єкта ***WshShortcut*** описані в табл. 2.15.

Табл. 2.15

Властивості об’єкта ***WshShortcut***

Властивість	**Опис**
Arguments	Містить рядок, що задає параметри командного рядка для ярлика
Description	Містить опис ярлика
FullName	Містить рядок з повним шляхом до ярлику

Закінчення табл.

HotKey	Задає гарячу клавішу для ярлика, тобто визначає комбінацію клавіш, за допомогою яких можна запустити або зробити активною програму, на яку вказує заданий ярлик
IconLocation	Задає шлях до іконки ярлика
TargetPath	Встановлює шлях до файлу, на який вказує ярлик
WindowStyle	Визначає вигляд вікна для додатка, на який вказує ярлик
WorkingDirectory	Задає робочий каталог для додатка, на який вказує ярлик

У наступному прикладі створюється ярлик для поточного сценарію з двома параметрами:

```
var WshShell = WScript.CreateOb]ect("WScript.Shell");
var oShellLink = WshShell.CreateShortcut("Current Script.Ink");
oShellLmk.TargetPath = WScript. ScriptFullName;
oShellLink.Arguments = "-a abc.txt";
oShellLink.Save();
```

Для того, щоб призначити ярлику гарячу клавішу, необхідно у властивість ***HotKey*** записати рядок, що містить назви потрібних клавіш, розділені символом "+".

У наступному прикладі на робочому столі створюється ярлик для Блокнота, якому призначається гаряча клавіша (комбінація **<Ctrl>+<Alt>+<D>**):

```
var WshShell = WScript.CreateOrrject("WScript.Shell");
strDesktop = WshShell.SpecialFolders("Desktop");
oMyShortcut = WshShell.CreateShortcut(strDesktop+"\\a_key.lnk");
oMyShortcut.TargetPath = "%wmdir%\\notepad.exe";
oMyShortcut.Hotkey = "CTRL+ALT+D";
oMyShortcut.Save();
WScript.Echo(oMyShortcut.Hotkey);
```

2.3.4. Об’єкти-колекції

У WSH входять об’єкти, за допомогою яких можна отримати доступ до колекцій, що містить такі елементи:

- параметри командного рядка запущеного сценарію або ярлика Windows (об’єкти ***WshArguments***, ***WshNamed*** і ***WshUnnamed***);
- значення змінних середовища (об’єкт ***WshEnvironment***);
- шляху до спеціальних папках Windows (об’єкт ***WshSpecial Folders***).

Об'єкт WshArguments

Об'єкт ***WshArguments*** містить колекцію всіх параметрів командного рядка запущеного сценарію або ярлика Windows. Цей об'єкт можна створити тільки за допомогою властивості ***Arguments*** об'єктів ***WScript*** і ***WshShortcut***.

За допомогою об'єкта ***WshArguments*** можна також виділяти і окремо обробляти аргументи сценарію, у яких є імена (наприклад, ***/Name:Andrey***) і безіменні аргументи. Зрозуміло, що використання іменних параметрів зручніше, оскільки в цьому випадку немає необхідності запам'ятовувати, в якому порядку повинні бути записані параметри під час запуску того чи іншого сценарію.

Для доступу до іменних і безіменним аргументів використовуються відповідно два спеціальних властивості об'єкта ***WshArguments***: ***Named*** і ***Unnamed***. Властивість ***Named*** містить посилання на колекцію ***WshNamed***, властивість ***Unnamed*** – на колекцію ***WshUnnamed***.

Таким чином, обробляти параметри командного рядка запущеного сценарію можна трьома способами:

- переглядати повний набір всіх параметрів (як іменних, так і безіменних) за допомогою колекції ***WshArguments***;
- виділити тільки ті параметри, які мають імена (іменні параметри) за допомогою колекції ***WshNamed***;
- виділити тільки ті параметри, які не мають імен (безіменні параметри) за допомогою колекції ***WshUnnamed***.

Далі наведений приклад сценарію, в якому на екран виводяться загальна кількість параметрів командного рядка, кількість іменних і безіменних аргументів, а також значення кожної з цих груп параметрів.

```
var i, Arg, objArgs, s, objNamedArgs, objUnnamedArgs; //Оголошуємо змінні
var objArgs = WScript.Arguments; //Створюємо об'єкт WshArguments
//Визначаємо загальну кількість аргументів
s = "Total arguments:" + objArgs.Count() + "\n";

for (var i=0; i<objArgs.length; i++) {
    s += objArgs(i) + "\n";
}

objUnnamedArgs = objArgs.Unnamed //Створюємо об'єкт WshUnnamed
//Визначаємо кількість безіменних аргументів
s += "\nUnnamed arguments:" + objUnnamedArgs.length + "\n";
for (var i=0; i<objUnnamedArgs.length; i++) {
    s += objUnnamedArgs(i) + "\n";
}
```

```
objNamedArgs = objArgs.Named; //Створюємо об'єкт WshNamed
//Визначаємо кількість іменних аргументів
s += "\nNamed arguments:" + objNamedArgs.length + "\n";
//Перевіряємо, чи існує аргумент /Ім'я:
if (objNamedArgs.Exists("Name"))
  s += objNamedArgs("Name") + "\n";

//Перевіряємо, чи існує аргумент /Comp:
if (objNamedArgs.Exists("Comp"))
  s +=objNamedArgs ("Comp") + "\n";

WScript.Echo(s); //Виводимо сформовані рядки
```

2.3.5. Об'єкт WshEnvironment

Об'єкт ***WshEnvironment*** дозволяє отримати доступ до колекції, що містить змінні середовища заданого типу (змінні середовища операційної системи, змінні середовища користувача або змінні середовища поточного командного вікна). Цей об'єкт можна створити за допомогою властивості Environment об'єкта ***WshShell*** або однойменного його методу:

```
WshShell = WScript.CreateObject("WScript.Shell");
WshSysEnv = WshShell.Environment;
WshUserEnv = WshShell.Environment("User");
```

Об'єкт ***WshEnvironment*** має властивість ***Length***, в якому зберігається число елементів у колекції (кількість змінних середовища), і методи ***Count*** і ***Item***. Для того, щоб отримати значення певної змінної середовища, як аргумент методу ***Item*** вказується ім'я цієї змінної в подвійних лапках. У наступному прикладі ми виводимо на екран значення змінної середовища ***PATH***.

```
var WshShell, WshSysEnv;
WshShell = WScript.CreateObject("WScript.Shell");
WshSysEnv = WshShell.Environment;
WScript.Echo("System Path: ",WshSysEnv.Item("PATH"));
```

Можна також просто вказати ім'я змінної в круглих дужках після імені об'єкта:

```
WScript.Echo("Системний шлях: ", WshSysEnv("PATH"));
```

Крім цього, в об'єкта ***WshEnvironment*** є метод ***Remove(strName)***, який видаляє задану змінну середовища.

2.3.6. Об'єкт WshSpecialFolders

Під час установки Windows завжди автоматично створюються кілька спеціальних папок (наприклад, папка для робочого столу (Desktop) або папка для меню Пуск (Start)), шлях до яких згодом може бути тим чи іншим способом змінений. Об'єкт WshSpecialFolders забезпечує доступ до колекції, що містить шляху до спеціальних папках Windows; завдання шляхів до таких папок може знадобитися, наприклад, для створення безпосередньо зі сценарію ярликів на робочому столі.

Об'єкт ***WshSpecialFolders*** створюється за допомогою властивості ***SpecialFolders*** об'єкта ***WshShell***:

```
WshShell = WScript.CreateObject("WScript.Shell");
WshSpecFold = WshShell.SpecialFolders;
```

Далі наведено сценарій, який формує список всіх наявних у системі спеціальних папок.

```
var WshShell, WshFldrs, SpecFldr, s; //Оголошуємо змінні
//Створюємо об'єкт WshShell
WshShell = WScript.CreateObject("Wscript.Shell");
//Створюємо об'єкт WshSpecialFolders
WshFldrs = WshShell.SpecialFolders;
s = "List of all special folders: " + "\n\n";
//Перебираємо всі елементи колекції WshFldrs

for (var i=0; i<WshFldrs.length; i++) {
    s+=WshFldrs(i)+"\n"; //Формуємо рядок зі значеннями аргументів
}

WScript.Echo(s);
```

2.4. Сценарії WSH для доступу до файлової системи. Об'єктна модель FileSystemObject

Сценарії JScript або VBScript, вбудовані в HTML-сторінки, можуть забороняти певні операції залежно від рівня безпеки, встановленого в налаштуваннях браузера. На відміну від таких сценаріїв, сценарії WSH надають повний доступ до файлової системи комп'ютера.

Існує 8 об'єктів для використання файлової системи зі сценаріїв WSH, головним з яких є ***FileSystemObject***. За допомогою методів цього об'єкта можна виконати такі основні дії:

– копіювання або переміщення файлів та каталогів;
– видалення файлів та каталогів;
– створення каталогів;
– створення або відкриття текстових файлів;
– створення об'єктів ***Drive***, ***Folder*** і ***File*** для доступу до певних дисків, каталогів або файлів відповідно.

Ви можете використовувати властивості об’єктів ***Drive***, ***Folder*** і ***File***, щоб отримати детальну інформацію про елементи файлової системи, пов’язані з ними. Об’єкти ***Folder*** і ***File*** також забезпечують методи обробки файлів і каталогів (створення, видалення, копіювання, переміщення); ці методи в основному копіюють відповідні методи об’єкта ***FileSystemObject***.

Крім того, існує три колекції об’єктів: ***Drives***, ***Folders*** і ***Files***. Колекція ***Drives*** містить об’єкти ***Drive*** для всіх доступних дисків у системі, ***Folders*** – об’єкти ***Folder*** для всіх підкаталогів цього каталогу та ***Files*** – об’єкти ***File*** для всіх файлів у певному каталозі.

Нарешті, ви можете прочитати інформацію з текстового файлу сценарію і записати в нього дані. Об’єкт ***TextStream*** надає методи для цієї мети.

У табл. 2.16 стисло описано, які об’єкти, атрибути та методи можуть знадобитися для виконання найбільш часто використовуваних файлових операцій.

Табл. 2.16

Виконання основних файлових сценаріїв

Операція	**Використовувані об’єкти, властивості і методи**
Отримання відомостей про певний диск (тип файлової системи, мітка тому, загальний обсяг і кількість вільного місця та ін.)	Властивості об’єкта *Drive*. Сам об’єкт *Drive* створюється за допомогою методу *GetDrive* об’єкта *FileSystemObject*
Отримання відомостей про заданий каталог або файл (дата створення або останнього доступу, розмір, атрибути та ін.)	Властивості об’єктів *Folder* і *File*. Самі ці об’єкти створюються за допомогою методів *GetFolder* і *GetFile* об’єкта *FileSystemObject*
Перевірка існування певного диска, каталогу або файлу	Методи *DriveExists*, *FolderExists* і *FileExists* об’єкта *FileSystemObject*
Копіювання файлів і каталогів	Методи *CopyFile* і *CopyFolder* об’єкта *FileSystemObject*, а також методи *File.Copy* і *Folder.Copy*
Переміщення файлів і каталогів	Методи *MoveFile* і *MoveFolder* об’єкта *FileSystemObject*, або методи *File.Move* і *Folder.Move*
Видалення файлів і каталогів	Методи *DeleteFile* і *DeleteFolder* об’єкта *FileSystemObject*, або методи *File.Delete* і *Folder.Delete*
Створення каталогу	Методи *FileSystemObject.CreateFolder* або *Folders.Add*

Закінчення табл.

Створення текстового файлу	Методи *FileSystemObject.CreateTextFile* або *Folder.CreateTextFile*
Отримання списку всіх доступних дисків	Колекція *Drives*, що міститься у властивості *FileSystemObject.Drives*
Отримання списку всіх підкаталогів заданого каталогу	Колекція *Folders*, що міститься у властивості *Folder.SubFolders*
Отримання списку всіх файлів заданого каталогу	Колекція *Files*, що міститься у властивості *Folder.Files*
Відкриття текстового файлу для читання, запису або додавання	Методи *FileSystemObject.CreateTextFile* або *File.OpenAsTextStream*
Читання інформації з заданого текстового файлу або запис її в нього	Методи об’єкта *TextStream*

2.4.1. Об’єкт FileSystemObject

Об’єкт ***FileSystemObject*** є основним об’єктом, що забезпечує доступ до файлової системи комп’ютера, а його методи використовуються для створення інших об’єктів (***Drives***, ***Driv***e, ***Folders***, ***Folder***, ***Files***, ***File*** і ***Textstream***).

Щоб створити об’єкт ***FileSystemObject*** у сценарії, ви можете використовувати метод ***CreateObject*** об’єкта ***WScript***:

```
var fso = WScript.CreateObject("Scripting.FileSystemObject");
```

Ви також можете використовувати об’єкт ***ActiveXObject*** мови ***JScript*** (можна використовувати цей об’єкт для роботи з файловою системою через сценарії на HTML-сторінках):

```
var fso = new ActiveXObject("Scripting.FileSystemObject");
```

Об’єкт ***FileSystemObject*** має єдину властивість ***Drives***, яка зберігає колекцію, що містить об’єкти ***Drive*** для всіх доступних дисків. У табл. 2.17 наведені методи об’єкта ***FileSystemObject***.

Табл. 2.17

Методи об’єкта ***FileSystemObject***

Метод	Опис
BuildPath(path, name)	Додає нове ім’я до вказаного шляху
CopyFile(source, destination [, overwrite])	Копіює один або кілька файлів з одного в інше розташування
CopyFolder(source, destination [, overwrite])	Копіює каталог, що містить усі підкаталоги, з одного в інше розташування

Продовження табл.

CreateFolder(foldername)	Створіть новий каталог. Якщо каталог з цим ім’ям вже існує, буде повідомлено про помилку
CreateTextFile(filename [, Overwrite [, Unicode]])	Створює новий текстовий файл та поверніть об’єкт *TextStream*, що вказує на нього
DeleteFile(filespec [, force])	Видаляє файл у шляху, вказаному параметром *filespec*
DeleteFolder(folderspec [, force])	Видаляє каталог, вказаний із параметром *folderspec*, і весь його вміст
DriveExists(drivespec)	Якщо пристрій, зазначений параметром *drivespec*, існує, повертає *true*, інакше повертає *false*
FileExists(filespec)	Якщо файл, зазначений параметром *filespec*, існує, він повертає *true*, інакше повертає *false*
FolderExists(folderspec)	Якщо вказаний каталог специфікацій папок існує, повертає *true*, інакше повертає *false*
GetAbsolutePathName(partspec)	Повертає повний шлях заданого відносного шляху *pathspec* (з поточного каталогу)
GetBaseName(path)	Повертає базове ім’я останнього компонента у шляху (без розширення)
GetDrive(drivespec)	Повертає об’єкт *Drive*, що відповідає диску, вказаному параметром *drivespec*
GetDriveName(path)	Повертає рядок, що містить ім’я диска у вказаному шляху. Якщо ім’я диска не вдається витягти з параметра *path*, метод повертає порожній рядок ("")
GetExtensionName(path)	Повертає рядок, що містить розширення останнього компонента у шляху. Якщо не можна вибрати компонент шляху з параметра *path*, метод поверне порожній рядок (""). Для мережевих дисків кореневий каталог (\) вважається частиною шляху
GetFile(filespec)	Повертає об’єкт *File*, що відповідає файлу, вказаному параметром *filespec*. Якщо файл (шлях, визначений параметром *filespec*) не існує, метод зазнає помилки
GetFileName(pathspec)	Повертає ім’я файлу, вказане в повному шляху до файлу. Якщо не можна вибрати ім’я файлу з параметра *pathspec*, метод повертає порожній рядок ("")
GetFolder(folderspec)	Повертає об’єкт *Folder*, що відповідає каталогу, вказаному параметром *folderspec*. Якщо каталогу не існує, під час виконання методу буде виникати помилка

Закінчення табл.

GetParentFolderName(path)	Повертає рядок, що містить ім'я батьківського каталогу останнього компонента у вказаному шляху. Якщо батьківський каталог неможливо вказати для останнього компонента у шляху, вказаному параметром *path*, метод повертає порожній рядок ("")
GetSpecialFolder(folderpec)	Повертає об'єкт *Folder* деяких спеціальних папок Windows, визначених параметром *folderspec*
GetTempName()	Повертає випадково сформоване ім'я файлу або каталогу, яке можна використовувати для операцій, для яких потрібні тимчасові файли або каталоги
MoveFile(source, destination)	Переміщує один або кілька файлів з одного розташування (параметр *source*) в інше розташування (параметр *destination*)
MoveFolder(source, destination)	Переміщує один або кілька каталогів з одного розташування (параметр *source*) в інше розташування (параметр *destination*)
OpenTextFile(filename (, iomode [, create [, format]]])	Відкриває вказаний текстовий файл і повертає об'єкт *TextStream*, щоб використовувати файл

2.4.2. Об'єкт Drive

Ви можете використовувати об'єкт ***Drive*** для доступу до властивостей локального або мережевого диска. Використовуйте метод ***GetDrive*** для створення об'єкта ***Drive***, як показано нижче:

```
var fso, d;
fso = WScript.CreateObject("Scripting.FileSystemObject");
d = fso.GetDrive("C");
```

Подібним чином об'єкт ***Drive*** можна отримати як елемент колекції ***Drives***. У табл. 2.18 перелічені властивості об'єкта ***Drive***, який не має методів.

Табл. 2.18

Властивості об'єкта ***Drive***

Властивість	Опис
AvailableSpace	Містить кількість користувацького простору, доступного на диску (у байтах)
DriveLetter	Містить літеру, пов'язану з локальним пристроєм або мережевим ресурсом. Ця властивість доступна лише для читання

Закінчення табл.

DriveType	Містить значення, яке визначає тип пристрою – 0 – невідомий пристрій; – 1 – пристрій зі знімним носієм; – 2 – жорсткий диск; – 3 – мережевий диск; – 4 – CD-ROM; – 5 – RAM-диск
FileSystem	Містить тип файлової системи, що використовується на диску (FAT, NTFS або CDFS)
FreeSpace	Містить кількість вільного місця (у байтах) на локальних дисках або мережевих ресурсах. Доступна лише для читання
IsReady	Якщо пристрій готовий, містить *true*, інакше – *false*. Для знімних носіїв та дисководів CD-ROM повертає значення *true* лише тоді, коли відповідний носій вставлений у дисковод і пристрій готовий надати доступ до носія.
Path	Містить шлях до диска (наприклад, *C*, але не *C*\)
RootFolder	Містить об’єкт *Folder*, що відповідає кореневому каталогу на диску. Доступна лише для читання
SerialNumber	Містить десятковий серійний номер тому заданого диска
ShareName	Містить мережеву назву диска. Якщо об’єкт не є мережевим диском, властивість містить порожній рядок ("")
TotalSize	Містить загальний обсяг у байтах локального диска або мережевого ресурсу
VolumeName	Містить мітку тому для диска. Доступна для читання і запису

Колекція для читання ***Drives*** містить об’єкти для всіх доступних комп’ютерних дисків (включаючи мережеві диски та знімні носії).

Властивість ***count*** колекції ***Drives*** зберігає кількість її елементів, тобто кількість доступних для використання дисків.

```
var FSO, DriveCol, d;
FSO = WScript.CreateObject("Scripting.FileSystemObject");
DriveCol = fso.Drives;
//Вилучення елемента колекції
d = DriveCol.Item("C:");
WScript.Echo(“Disk C has ",d.FreeSpace,"kb free");
```

Для перерахування всіх елементів колекції ***Drives*** потрібно використовувати об’єкт ***Enumerator***, який був описаний вище.

2.4.3. Об'єкт Folder

Об'єкт ***Folder*** забезпечує доступ до властивостей каталогу. Ви можете використовувати властивість ***RootFolder*** об'єкта ***Drive*** або методи ***GetFolder***, ***GetParentFolder*** та ***GetSpecialFolder*** об'єкта ***FileSystemObject*** для створення цього об'єкта таким чином:

```
var fso, folder;
fso = WScript.CreateObject("Scripting.FileSystemObject");
folder = fso.GetFolder ("C:\\My Documents");
```

Об'єкт ***Folder*** також можна отримати як елемент колекції ***Folders***.

Властивості об'єкта ***Folder*** показані в табл. 2.19.

Табл. 2.19

Властивості об'єкта ***Folder***.

Властивість	Опис
Attributes	Дозволяє переглядати або встановлювати атрибути каталогу
DateCreated	Містить дату та час створення каталогу. Доступна лише для читання
DateLastAccessed	Містить дату та час останнього доступу до каталогу. Доступна лише для читання
DateLastModified	Містить дату та час останньої зміни каталогу. Доступна лише для читання
Drive	Містить необхідну літеру пристрою, де знаходиться каталог. Доступна лише для читання
Files	Містить колекцію *Files*, що складається з об'єктів *File* усіх файлів у каталозі (включаючи приховані і системні)
IsRootFolder	Якщо каталог є кореневим, містить *true*, інакше – *false*
Name	Дозволяє переглядати та перейменовувати каталоги, доступна для читання та запису
ParentFolder	Містить об'єкт *Folder*, що містить батьківський каталог. Доступна лише для читання
Path	Містить шлях до каталогу
ShortName	Містить коротку назву каталогу
ShortPath	Містить шлях до каталогу, що складається з коротких імен каталогів
Size	Містить розмір (у байтах) усіх файлів та підкаталогів, що містяться в цьому каталозі
SubFolders	Містить колекцію Folders, що складається з усіх підкаталогів каталогу (включаючи підкаталоги з атрибутами "Прихований" і "Системний")
Type	Містить інформацію про тип каталогу

Методи об’єкта ***Folder*** описані в табл. 2.20.

Табл. 2.20

Методи об’єкта ***Folder***

Метод	Опис
Copy(destination [, overwrite])	Копіює каталог в інше місце
CreateTextFile(filename [,overwrite [,Unicode]])	Створює новий текстовий файл з ім’ям *filename* І повертає об’єкт TextStream, що вказує на цей файл (цей метод аналогічний розглянутому вище методу *CreateTextFile* об’єкта *FileSystemObject*)
Delete([force])	Видаляє каталог
Move(destination)	Переміщує каталог в інше місце

Колекція ***Folders*** містить об’єкти ***Folder*** для всіх підкаталогів видаленного каталогу. Створюється ця колекція з допомогою властивості ***SubFolders*** відповідного об’єкта ***Folder***. Наприклад, в наступному прикладі змінна ***fc*** є колекцією, що містить об’єкти ***Folder*** для всіх підкаталогів каталогу ***C:\Program Files***.

```
var fso, f, fc;
fso = WScript.CreateObject("Scripting FileSystemObject");
f = fso.GetFolder("C:\\Program Files");
fc = f .Sub Folders;
```

Колекція ***Folders*** (як і ***Drives***) має властивість ***Count*** та метод ***Item***. Крім цього, у ***Folders*** є метод ***Add(folderName)***, дозволяє створювати нові підкаталоги.

Для доступу до всіх елементів колекції потрібно використовувати, як зазвичай, об’єкт ***Enumerator***.

```
var fso, f, fc, s;
fso = WScript.CreateObject("Scripting.FileSystemObject");
f = fso.GetFolder("C:\\Program Files");
fc = new Enumerator(f.SubFolders);
s = "";
for (; 'fc.atEnd(); fc.moveNext())
{
    s += fc.item();
    s += "\n";
}
WScript.Echo(s);
```

2.4.4. Об'єкт File

Об'єкт ***File*** забезпечує доступ до всіх властивостей файлу. Створити цей об'єкт можна за допомогою методу ***GetFile*** об'єкта ***FileSystemObject*** таким чином:

```
var fso, f;
fso = WScript.CreateObject ("Scripting.FileSystemObject");
f = fso.GetFile("C:\\My Documents\\letter.txt");
```

Також об'єкти ***File*** можуть бути отримані як елементи колекції ***Files***. Властивості об'єкта ***File*** описані в табл. 2.21.

Табл. 2.21

Властивості об'єкта ***File***

Властивість	Опис
Attributes	Дозволяє переглянути або встановити атрибути файлів
DateCreated	Містить дату та час створення файлу. Доступно тільки для читання
DateLastAccessed	Містить дату та час останнього доступу до файлу. Доступно тільки для читання
DateLastModffied	Містить дату та час останньої модифікації файлу. Доступно тільки для читання
Drive	Містить букву диска для пристрою, на якому знаходиться файл. Доступно тільки для читання
Name	Дозволяє переглянути і змінити ім'я файлу. Доступно для читання і запису
ParentFolder	Містить об'єкт *Folder* для батьківського каталогу файла. Доступно тільки для читання
Path	Містить шлях до файлу
ShortName	Містить коротку назву файлу
ShortPath	Містить шлях до файлу, що складається з коротких імен каталогів
Size	Містить розмір заданого файлу в байтах
Type	Повертає інформацію про тип файлу. Наприклад, для файлу з розширенням *txt* повернеться рядок *Text Document*

Методи об'єкта ***File*** представлені в табл. 2.22.

Табл. 2.22

Методи об'єкта ***File***

Метод	Опис
Copy(destination [, overwrite])	Копіює файл в інше місце
Delete([force])	Видаляє файл
Move(destination)	Переміщує файл в інше місце
OpenAsTextStream([iomode, [format]])	Відкриває заданий файл і повертає об'єкт *TextStream*, який може бути використаний для читання, запису або додавання даних в текстовий файл

Колекція ***Files*** містить об’єкти File для всіх файлів, що знаходяться всередині певного каталогу. Створюється ця колекція з допомогою властивості ***Files*** відповідного об’єкта ***Folder***. Для доступу в циклі до всіх елементів колекції ***Files*** використовується об’єкт ***Enumerator***.

```
var fso, f, fl, fc, s;
fso = WScript.CreateObject("Scripting.FileSystemObject");
f = fso.GetFolder("C:\\My Documents");
fc = new Enumerator(f.files),
for (; !fc.atEnd(); fc.moveNext())
{
    s += fc.item();
    s += "\n";
}
WScript.Echo(s);
```

2.4.5. Приклади сценаріїв

Далі наведені прості приклади сценаріїв, які працюють з файловою системою (створення, копіювання, видалення файлів і каталогів, читання і запис рядків у текстовому файлі та ін.).

Отримання відомостей про диск

Доступ до властивостей заданого локального або мережевого диска можна отримати за допомогою об’єкта ***Drive***, який повертається методом ***GetDrive*** об’єкта ***FileSystemObject***, а також може бути отриманий як елемент колекції ***Drives***.

Далі наведений сценарій ***DriveInfo.js***, який виводить на екран деякі властивості диска ***C***.

```
//Оголошуємо змінні
var FSO, d, TotalSize, FreeSpace, s;
//Створюємо об'єкт FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Створюємо об'єкт Drive для диска C
d = FSO.GetDrive("C:");
s = "Information about drive C:\n";
//Отримуємо серійний номер диска
s += "Serial number: " + d.SerialNumber + "\n";
//Отримуємо мітку тому диска
s += "Label: " + d.VolumeName + "\n";
//Обчислюємо загальний обсяг диска в кілобайтах
TotalSize = d.TotalSize/1024;
s += "Total: " + TotalSize + " Kb\n";
//Обчислюємо обсяг вільного простору диска в кілобайтах
FreeSpace = d.FreeSpace/1024;
s += "Free: " + FreeSpace + " Kb\n";
//Виводимо властивості диска на екран
WScript.Echo(s);
```

Отримання відомостей про каталог

Доступ до властивостей каталогу забезпечує об'єкт ***Folder***. Створити цей об'єкт можна за допомогою властивості ***RootFolder*** об'єкта ***Drive*** або методів ***GetFolder***, ***GetParentFolder*** і ***GetSpecial Folder*** об'єкта ***FileSystemObject***. Також об'єкти ***Folder*** можуть бути отримані як елементи колекції ***Folders***.

У сценарії ***FolderInfo.js*** на екран виводяться властивості каталогу, з якого був запущений сценарій.

```
var FSO, WshShell, FoldSize, s; //Оголошуємо змінні

//Створюємо об'єкт FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Створюємо об'єкт WshShell
WshShell = WScript.CreateObject("WScript.Shell");

//Визначаємо каталог, з якого був запущений сценарій
//(Поточний каталог)
Folder = FSO.GetFolder(WshShell.CurrentDirectory);
//Отримуємо ім'я поточного каталогу
s = "Current directory: " + Folder.Name + "\n";
//Отримуємо дату створення поточного каталогу
s += "Creation date: " + Folder.DateCreated + "\n";
//Обчислюємо розмір поточного каталогу в кілобайтах
FoldSize = Folder.Size/1024;
s += "Total size: " + FoldSize + " Kb\n";
//Виводимо інформацію на екран
WScript.Echo(s);
```

Отримання інформації про файл

Доступ до всіх властивостей файлу забезпечує об'єкт ***File***, створити який можна за допомогою колекції ***Files*** або методу ***GetFile*** об'єкта ***FileSystemObject***.

Далі наведений сценарій ***FileInfo.js***, в якому на екран виводяться деякі властивості файлу ***C:\mtsearchdump.txt***.

```
var FSO, f, s //Оголошуємо змінні
//Створюємо об'єкт FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject")
//Створюємо об'єкт File
f = FSO.GetFile("C:\\mtsearchdump.txt")

//Отримуємо ім'я файлу
s = "File: " + f.Name + "\n";
//Отримуємо дату створення файлу
s += "Creation date: " + f.DateCreated + "\n";
//Отримуємо тип файлу
s += "Type: " + f.Type + "\n";
//Виводимо інформацію на екран
WScript.Echo(s);
```

Створення каталогу

Створити новий каталог на диску можна або за допомогою методу ***CreateFolder*** об'єкта ***FileSystemObject*** або за допомогою методу ***Add*** колекції ***Folders***. Обидва ці методи використовуються в сценарії ***MakeFolder.js*** для створення на диску ***N*** каталогу ***New folder*** та підкаталогу ***Another folder*** у ньому.

```
//Оголошуємо змінні
var FSO, f, SubFolders;

//Створюємо об'єкт FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Створюємо каталог N:\NewFolder
FSO.CreateFolder("N:\\New folder");
//Створюємо об'єкт Folder для каталогу N:\NewFolder
f = FSO.GetFolder("N:\\New folder");
//Створюємо колекцію підкаталогів каталогу N:\NewFolder
SubFolders = f.SubFolders;
//Створюємо каталог N:\New folder\Another folder
SubFolders.Add("Another folder");
```

Створення текстового файлу

Для створення текстового файлу використовується метод ***Create TextFile*** об'єкта ***FileSystemObject***, який має один обов'язковий текстовий параметр (шлях до створюваного файлу) і два необов'язкових логічних параметри (***Overwrite*** і ***Unicode***).

Параметр ***Overwrite*** має значення в тому випадку, коли створюється файл, що вже існує. Якщо ***Overwrite*** дорівнює ***True***, то такий файл перепишеться (старий вміст буде втрачений), якщо ж як ***Overwrite*** вказано ***False***, то файл перезаписаний не буде. Якщо цей параметр взагалі не вказано, то існуючий файл також не буде переписаний.

Параметр Unicode вказує, в якому форматі (ASCII або Unicode) слід створювати файл. Якщо Unicode дорівнює True, то файл створюється в форматі Unicode, якщо ж Unicode дорівнює False або цей параметр взагалі не вказано, то файл створюється в режимі ASCII.

У сценарії ***CreateTempFile.js*** показано, яким чином можна створити файл з випадково обраним ім'ям (такі файли часто використовуються для запису тимчасових даних).

```
var FSO, FileName, f, s; //Оголошуємо змінні
//Створюємо об'єкт FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Генеруємо випадкове ім'я файлу
FileName = FSO.GetTempName();
//Створюємо файл з ім'ям FileName
f = FSO.CreateTextFile(FileName, true);
//Закриваємо файл
f.Close();
//Повідомляємо про створення файлу
WScript.Echo("File " + FileName + " is created");
```

Копіювання і переміщення файлів і каталогів

Для копіювання файлів / каталогів можна застосовувати метод ***CopyFile/CopyFolder*** об’єкта ***FileSystemObject*** або метод ***Copy*** відповідного до цього файлу / каталогу об’єкта ***File/Folder***. Переміщаються файли / каталоги за допомогою методів ***MoveFile/Move Folder*** об’єкта ***FileSystemObject*** або методу ***Move*** відповідного до цього файлу / каталогу об’єкта ***File/Folder***.

Зазначимо, що у разі використання всіх цих методів процес копіювання або переміщення переривається після першої помилки, що виникла. Крім того, не можна переміщати файли і каталоги з одного диска на інший.

Далі наведено сценарій ***CopyFile.js***, який ілюструє використання методу ***Copy***. У цьому сценарії на диску ***C*** створюється файл ***TestFile.txt***, який потім копіюється на робочий стіл.

```
//Оголошуємо змінні
var FSO, f, WshShell, WshFldrs, PathCopy;

//Створюємо об'єкт FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Створюємо файл
f = FSO.CreateTextFile("N:\\TestFile.txt",true);
//Записуємо в файл рядок
f.WriteLine("Test file");
//Закриваємо файл
f.Close();

//Створюємо об'єкт WshShell
WshShell = WScript.CreateObject("WScript.Shell");
//Створюємо об'єкт WshSpecialFolders
```

```
WshFldrs = WshShell.SpecialFolders;
//Визначаємо шлях до робочого столу
PathCopy = WshFldrs.item("Desktop") + "\\";
//Створюємо об'єкт File для файлу N:\TestFile.txt
f = FSO.GetFile("N:\\TestFile.txt");
//Копіюємо файл на робочий стіл
f.Copy(PathCopy);
```

Видалення файлів і каталогів

Для видалення файлів / каталогів можна застосовувати метод ***DeleteFile/DeleteFolder*** об’єкта ***FileSystemObject*** або метод ***Delete*** відповідного до цього файлу / каталогу об’єкта ***File/Folder***. Зазначимо, що у разі видалення каталогу неважливо, чи є він порожнім, чи ні – видалення буде вироблено в будь-якому випадку. Якщо ж заданий для видалення файл/каталог не буде знайдений, то виникне помилка.

Далі наведено сценарій ***DeleteFile.js***, в якому проводиться видалення попередньо створеного файлу ***C:\TestFile.txt***.

```
var FSO, f, FileName;

//Створюємо об'єкт FileSystemObject
FSO = WScript.CreateObject("Scripting.FileSystemObject");
//Задаємо ім'я файлу
FileName = "N:\\TestFile.txt";
//Створюємо файл
f = FSO.CreateTextFile(FileName, true);
//Записуємо в файл рядок
f.WriteLine("Test file");
//Закриваємо файл
f.Close();
WScript.Echo("File is created");
FSO.DeleteFile(FileName);
WScript.Echo("File is deleted");
```

РОЗДІЛ 3. КОМАНДНА ОБОЛОНКА WINDOWS POWERSHELL

Нова оболонка ***Windows PowerShell*** (попередньо вона була названа ***Monad***) була задумана розробниками ***Microsoft*** як більш потужне середовище для написання сценаріїв та роботи з командного рядка. Розробники ***PowerShell*** переслідували кілька цілей. Головна і найамбітніша з них – створити середовище складання сценаріїв, яке найкращим чином підходило б для сучасних версій операційної системи Windows і було б більш функціональним, що розширюється і є простим у використанні, ніж будь-який аналогічний продукт для будь-якої іншої операційної системи. У першу чергу, це середовище повинно було підходити для вирішення завдань, що стоять перед системними адміністраторами (тим самим Windows отримала б додаткову перевагу в боротьбі за сектор корпоративних платформ), а також задовольняти вимоги розробників програмного забезпечення, надаючи їм засоби для швидкої реалізації інтерфейсів управління до створюваних додатків.

Для досягнення цих цілей були вирішені зазначені завдання.

Забезпечення прямого доступу з командного рядка до об’єктів ***COM***, ***WMI*** і ***.NET***. У новій оболонці присутні команди, що дозволяють в інтерактивному режимі працювати з COM-об’єктами, а також з екземплярами класів, визначених в інформаційних схемах ***WMI*** і ***.NET***.

Організація роботи з довільними джерелами даних у командному рядку за принципом файлової системи. Наприклад, навігація системним реєстром або сховищем цифрових сертифікатів виконується з командного рядка за допомогою аналога команди CD інтерпретатора ***Cmd.exe***.

Розробка інтуїтивно зрозумілої уніфікованої структури вбудованих команд, заснованої на їх функціональному призначенні. У новій оболонці імена всіх внутрішніх команд (в ***PowerShell*** вони називаються Командлети) відповідають шаблону «дієслово-іменник», наприклад, ***Get-Process*** (отримати інформацію про процес), ***Stop-Service*** (зупинити службу), ***Clear-Host*** (очистити екран консолі) та ін. Для однакових параметрів внутрішніх команд використовуються стандартні імена, структура параметрів у всіх командах ідентична, всі

команди обробляються одним синтаксичним аналізатором. У результаті полегшується запам’ятовування і вивчення команд.

Забезпечення можливості розширення вбудованого набору команд. Внутрішні команди ***PowerShell*** можуть доповнюватися командами, які створюються користувачем. При цьому вони повністю інтегруються в оболонку, інформація про них може бути отримана зі стандартної довідкової системи ***PowerShell***.

Організація підтримки знайомих команд з інших оболонок. У ***PowerShell*** на рівні псевдонімів власних внутрішніх команд підтримуються найбільш часто використовувані стандартні команди з оболонки ***Cmd.exe*** і Unix-оболонок. Наприклад, якщо користувач, який звик працювати з Unix-оболонкою, виконає ***ls***, то він отримає очікуваний результат: список файлів у поточному каталозі (те ж саме стосується команди ***dir***).

Розробка повноцінної вбудованої довідкової системи для внутрішніх команд. Для більшості внутрішніх команд в довідковій системі подано докладний опис і приклади використання. У будь-якому випадку вбудована довідка за будь-якою внутрішньою командою буде містити короткий опис всіх її параметрів.

Реалізація автоматичного завершення у разі введення з клавіатури імен команд, їх параметрів, а також імен файлів і папок. Ця можливість значно спрощує і прискорює введення команд з клавіатури.

Головною особливістю середовища ***PowerShell***, що відрізняє її від всіх інших оболонок командного рядка, є те, що одиницею обробки і передачі інформації тут є об’єкт, а не рядок тексту.

Під час розробки будь-якої мови програмування одним з основних є питання про те, які типи даних і яким чином будуть у ньому представлені. У процесі створення ***PowerShell*** розробники вирішили не винаходити нічого нового і скористатися уніфікованою об’єктною моделлю ***.NET***. Цей вибір був зроблений з кількох причин.

По-перше, платформа ***.NET*** повсюдно використовується у розробці програмного забезпечення для Windows і представляє, зокрема, загальну інформаційну схему, за допомогою якої різні компоненти операційної системи можуть обмінюватися даними один з одним.

По-друге, об’єктна модель ***.NET*** є такою, що самодокументується: кожен об’єкт ***.NET*** містить інформацію про свою структуру. У разі інтерактивної роботи це дуже корисно, тому що з’являється можливість безпосередньо з командного рядка виконати запит до певного об’єкта і побачити опис його властивостей і методів, тобто зрозуміти, які саме маніпуляції можна виконати з цим об’єктом, не вивчаючи додаткової документації з його описом.

По-третє, працюючи в оболонці з об'єктами, можна за допомогою їхніх властивостей і методів легко отримувати потрібні дані, не займаючись розбором і аналізом символьної інформації, як це відбувається у всіх традиційних тексто-орієнтованих оболонках командного рядка. Розглянемо приклад. У Windows XP є консольна утиліта ***tasklist.exe***, яка видає інформацію про процеси, запущені у системі:

C:\> tasklist

Припустимо, що ми в командному файлі інтерпретатора ***Cmd.exe*** за допомогою цієї утиліти хочемо визначити, скільки оперативної пам'яті витрачає процес ***kavsvc.exe***. Для цього потрібно виділити з вихідного потоку команди ***tasklist*** відповідний рядок, витягти з нього підрядок, що містить необхідну кількість, і прибрати проміжки між розрядами (при цьому слід врахувати, що залежно від налаштувань операційної системи роздільником розрядів може бути не проміжок, а інший символ). У ***PowerShell*** аналогічна задача вирішується за допомогою команди ***get-process***, яка повертає колекцію об'єктів, кожен з яких відповідає одному запущеному процесу. Для визначення пам'яті, що витрачається процесом ***kavsvc.exe***, немає необхідності в додаткових маніпуляціях з текстом, досить просто взяти значення властивості WS об'єкта, яке відповідає такому процесу.

Нарешті, об'єктна модель ***.NET*** дозволяє ***PowerShell*** безпосередньо використовувати функціональність різних бібліотек, які є частиною платформи ***.NET***. Наприклад, щоб дізнатися, яким днем тижня було 9 листопада 1974 року, в ***PowerShell*** можна виконати таку команду:

```
PS F:\> (Get-date "09.11.1974").DayOfWeek
Saturday
```

У цьому випадку команда ***get-date*** повертає .NET-об'єкт ***DateTime***, що має властивість, у разі звернення до якого обчислюється день тижня для відповідної дати. Таким чином, розробникам ***PowerShell*** не потрібно створювати спеціальну бібліотеку для роботи з датами і часом – вони просто беруть готове рішення в ***.NET***.

Встановивши оболонку в системі, можна почати новий інтерактивний сеанс. Для цього слід натиснути на кнопку ***Пуск (Start)***, відкрити меню ***Всі програми (All Programs)*** і вибрати елемент ***Windows PowerShell***. Інший варіант запуску оболонки – пункт ***Виконати ... (Run)*** в меню ***Пуск (Start)***, ввести ім'я файлу ***powershell*** і натиснути кнопку OK.

У результаті відкриється нове командне вікно із запрошенням вводити команди.

Виконаємо першу команду в ***PowerShell***. Нехай це буде щось знайоме, наприклад, ***dir*** (команди в ***PowerShell*** обробляються без урахування регістру). На екран буде виведено список файлів у поточному каталозі:

```
PS F:\> dir


    Каталог: F:\


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
da----       14.04.2021     15:45                Backup
d-----       21.04.2021     12:17                dir1
d-----       21.04.2021     12:17                dir2
d-----       20.01.2021      1:44                Games
d-----       19.01.2021     23:21                Microsoft Flight Simulator
d-----       20.01.2021      0:36                Open Server
```

3.1. Типи команд PowerShell

В оболонці ***PowerShell*** підтримуються команди чотирьох типів: командлети, функції, сценарії і зовнішні виконувані файли.

Перший тип – так звані ***командлети (cmdlet)***. Цей термін використовується поки тільки всередині ***PowerShell***. Командлет є класом ***.NET***, породжений від базового класу ***Cmdlet***; розробляються командлети за допомогою пакета ***PowerShell Software Developers Kit (SDK)***. Єдиний базовий клас ***Cmdlet*** гарантує сумісний синтаксис всіх командлетів, а також автоматизує аналіз параметрів командного рядка і опис синтаксису командлетів для вбудованої довідки.

Такий тип команд компілюється в динамічну бібліотеку (DLL) і підвантажується до процесу ***PowerShell*** під час запуску оболонки (тобто самі по собі командлети не можуть бути запущені як додатки, але в них містяться виконувані об'єкти). Оскільки код, що компілюється, підвантажується до процесу оболонки, цей тип команд виконується найбільш ефективно. Командлети – це аналог внутрішніх команд традиційних оболонок.

Наступний тип команд – функції. Функція – це блок коду мовою ***PowerShell***, що має назву і знаходиться в пам'яті до завершення поточного сеансу командної оболонки. Функції, як і командлети, підтримують іменовані параметри. Аналіз синтаксису функції проводиться один раз під час її оголошення.

Сценарій – це блок коду мовою ***PowerShell***, що зберігається в зовнішньому файлі з розширенням ***ps1***. Аналіз синтаксису сценарію проводиться з кожним його запуском.

Останній тип команд – зовнішні виконувані файли, які виконуються звичайним чином операційною системою.

3.1.1. Імена та структура командлетів

Як уже було зазначено вище, в ***PowerShell*** аналогом внутрішніх команд є командлети. Командлети можуть бути дуже простими або дуже складними, але кожен з них розробляється для вирішення однієї, вузької задачі. Робота з командлетами стає по-справжньому ефективною під час використання їх композиції (конвеєризації об’єктів між командлетами).

Імена командлетів завжди відповідають шаблону «дієслово-іменник», де дієслово задає певну дію, а іменник визначає об’єкт, над яким цю дію буде здійснено.

Наприклад, ***Get-Process*** (отримати інформацію про процес), ***Stop-Service*** (зупинити службу), ***Clear-Host*** (очистити екран консолі) та ін. Щоб переглянути список командлетів, доступних у ході поточного сеансу, потрібно виконати командлет ***Get-Command***:

```
PS F:\> Get-Command

CommandType     Name                                       Version    Source
-----------     ----                                       -------    ------
Alias           Add-AdlAnalyticsDataSource                 4.2.3      AzureRM.DataLakeAnalytics
Alias           Add-AdlAnalyticsFirewallRule               4.2.3      AzureRM.DataLakeAnalytics
Alias           Add-AdlStoreFirewallRule                   5.2.0      AzureRM.DataLakeStore
Alias           Add-AdlStoreItemContent                    5.2.0      AzureRM.DataLakeStore
Alias           Add-AdlStoreTrustedIdProvider              5.2.0      AzureRM.DataLakeStore
Alias           Add-AzureHDInsightConfigValues             5.1.2      Azure
Alias           Add-AzureHDInsightMetastore                5.1.2      Azure
Alias           Add-AzureHDInsightStorage                  5.1.2      Azure
Alias           Add-AzureRmAccount                         4.6.0      AzureRM.Profile
Alias           Add-AzureRmIotHubEHCG                      3.1.2      AzureRM.IotHub
Alias           Add-ProvisionedAppxPackage                 3.0        Dism
Alias           Add-WAPackEnvironment                      5.1.2      Azure
Alias           Apply-WindowsUnattend                      3.0        Dism
Alias           Confirm-SSLegacyVolumeContainerStatus      5.1.2      Azure
Alias           Disable-AzureRmHDInsightOMS                4.1.2      AzureRM.HDInsight
Alias           Disable-AzureStorageSoftDelete             4.2.1      Azure.Storage
```

За замовчуванням командлет ***Get-Command*** інформує вас у трьох стовпцях: ***CommandType***, ***Name*** і ***Definition***. При цьому в стовпці ***Definition*** відображається синтаксис командлетів (три крапки (...) в стовпці синтаксису вказує на те, що дані обрізані).

Командлети можуть мати параметри, до яких можна звернутися на ім’я, перед яким ставиться дефіс (-), або за позицією (в останньому випадку інтерпретація параметра буде виконуватися залежно від його місця розташування в командному рядку).

Косі риски (/ і \) разом з параметрами в оболонці ***Windows PowerShell*** не використовуються.

У загальному випадку синтаксис командлетів має таку структуру:

Ім’я_командлета -параметр1 -параметр2 аргумент1 аргумент2

Тут ***параметр1*** – параметр (перемикач), що не має значення; ***параметр2*** – ім’я параметра, що має значення ***аргумент1***; ***аргумент2*** – параметр, який не має імені. Наприклад, командлет ***Get-Process*** має параметр ***Name***, який визначає ім’я процесу, інформацію про який потрібно вивести. Ім’я цього параметра вказувати необов’язково.

Таким чином, для отримання відомостей про процес ***Far*** можна ввести або команду ***Get-Process -Name Far***, або команду ***Get-Process Far***.

Перебуваючи в оболонці ***PowerShell***, можна ввести частину будь-якої команди, натиснути клавішу <***Tab***>, і система спробує сама завершити цю команду.

Подібне автоматичне завершення спрацьовує, по-перше, для імен файлів і шляхів файлової системи. У разі натискання клавіші <***Tab***> ***PowerShell*** автоматично розширить частково введений шлях файлової системи до першого знайденого збігу. У випадку повторення натискання клавіші <***Tab***> проводиться циклічний перехід за наявними можливостями вибору. Також у ***PowerShell*** реалізована можливість автоматичного завершення шляхів файлової системи на основі шаблонних символів (***?*** І *******). Наприклад, якщо ввести команду ***cd c:\pro* files*** і натиснути клавішу <***Tab***>, то в рядку введення з'явиться команда ***cd 'C:\Program Files'***.

По-друге, в ***PowerShell*** реалізовано автозавершення імен командлетів і їх параметрів. Якщо ввести першу частину імені командлет (дієслово) і дефіс, натиснути після цього клавішу <***Tab***>, то система підставить ім'я першого підходящого командлета (наступний підходящий варіант імені вибирається шляхом повторного натискання <***Tab***>). Аналогічним чином автозавершення спрацьовує для частково введених імен параметрів командлет: натискаючи клавішу <***Tab***>, ми будемо циклічно перебирати підходящі імена.

Нарешті, ***PowerShell*** дозволяє автоматично завершувати імена використовуваних змінних (об'єктів) та імена властивостей об'єктів.

3.1.2. Псевдоніми команд

Механізм псевдонімів, реалізований в оболонці ***PowerShell***, дає можливість користувачам виконувати команди за їх альтернативними іменами (наприклад, замість команди ***Get-Childitem*** можна користуватися псевдонімом ***dir***). У ***PowerShell*** заздалегідь визначено багато псевдонімів, можна також додавати власні псевдоніми в систему.

Псевдоніми в ***PowerShell*** діляться на два типи. Перший тип призначений для сумісності імен з різними інтерфейсами. Псевдоніми цього типу дозволяють користувачам, які мають досвід роботи з іншими оболонками (***Cmd.exe*** або ***Unix-оболонки***), використовувати знайомі їм імена команд для виконання аналогічних операцій у ***PowerShell***, що спрощує освоєння нової оболонки, дозволяючи не витрачати зусиль на запам'ятовування нових команд ***PowerShell***. Наприклад, користувач хоче очистити екран. Якщо у нього є досвід роботи з ***Cmd.exe***, то він, природно, спробує виконати команду ***cls***. ***PowerShell*** при цьому виконає командлет ***Clear-Host***, для якого ***cls*** є

псевдонімом і який виконує потрібні дії – очищення екрану. Для користувачів ***Cmd.exe*** в ***PowerShell*** визначені псевдоніми ***cd, cls, copy, del, dir, echo, erase, move, popd, pushd, ren, rmdir, sort, type***; для користувачів Unix – псевдоніми ***cat, chdir, clear, diff, h, history, kill, lp, ls, mount, ps, pwd, r, rm, sleep, tee, write***.

Дізнатися, який саме командлет ховається за знайомим псевдонімом, можна за допомогою командлета ***Get-Alias***:

```
PS F:\> Get-Alias cd

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Alias           cd -> Set-Location
```

Псевдоніми другого типу (стандартні псевдоніми) в ***PowerShell*** призначені для швидкого введення команд. Такі псевдоніми утворюються з імен командлетів, яким вони відповідають. Наприклад, дієслово ***Get*** скорочується до ***g***, дієслово ***Set*** скорочується до ***s***, іменник ***Location*** скорочується до ***l*** та ін. Таким чином, для командлета ***Set-Location*** відповідає псевдонім ***sl***, а командлета ***Get-Location*** – псевдонім ***gl***.

Переглянути список всіх пседонімов, оголошених у системі, можна за допомогою командлета ***Get-Alias*** без параметрів. Визначити власний псевдонім можна за допомогою командлета ***Set-Alias***.

3.1.3. Довідкова система PowerShell

У ***PowerShell*** передбачено кілька способів отримання довідкової інформації всередині оболонки.

Коротку довідку за кожним окремим командлетом можна отримати за допомогою параметра ***?*** (знак питання), зазначеного після імені цього командлета. Наприклад:

PS C:\> get-process -?

Більш детальну інформацію можна отримати за допомогою спеціального командлета Get-***Help*** з параметрами ***Detailed*** або ***Full*** (наприклад, ***Get-Help Get-Process -Full***). Запустивши цей командлет з параметром *, можна побачити всі доступні розділи довідкової системи.

Командлет ***Get-Help*** дозволяє переглядати довідкову інформацію не тільки про різні командлети, а й про синтаксис мови ***PowerShell***, про псевдоніми, про різні аспекти роботи оболонки функцій та ін. Список тем, обговорення яких представлено в довідковій службі ***PowerShell***, можна побачити в такий спосіб:

```
PS F:\> Get-Help about_*

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
about_BeforeEach_AfterEach        HelpFile                            performed at the beginning and end of every It...
about_Mocking                     HelpFile                            Pester provides a set of Mocking functions mak...
about_Pester                      HelpFile                            Pester is a BDD based test runner for PowerShell.
about_should                      HelpFile                            Provides assertion convenience methods for com...
about_TestDrive                   HelpFile                            A PSDrive for file activity limited to the sco...
```

Таким чином, щоб прочитати докладну інформацію щодо використання масивів у ***PowerShell***, потрібно виконати таку команду: ***Get-Help about_array***.

Командлет ***Get-Help*** виводить вміст розділу довідки на екран відразу цілком. Функції ***man*** і ***help*** дозволяють довідкову інформацію виводити поекранно (аналогічно команді ***MORE*** інтерпретатора ***Cmd.exe***), наприклад: ***man about_array***.

3.2. Робота з файловою системою у PowerShell

Навігація у файловій системі

В оболонці **cmd.exe** зміна поточного каталогу проводиться за допомогою команди ***cd***. У **PowerShell** команда ***cd*** має таке ж значення, при цьому вона є стандартним псевдонімом командлета ***Set-Location***. Інші псевдоніми – ***chdir***, ***sl***. Наприклад, наступна команда робить поточним каталог ***C:\Windows:***

PS C:\> cd C:\Windows
PS C:\Windows>

Як і в оболонці **cmd.exe** як шлях можна вказувати символи **..** (для переходу в батьківський каталог) і **** (для переходу в кореневий каталог поточного диску). При цьому потрібно враховувати такий нюанс. Людина, яка часто користувалась командою ***cd*** в оболонці ***cmd.exe***, швидше за все, буде автоматично набирати команди типу ***cd..*** або ***cd*** без додаткових проміжків. У **PowerShell** це викличе помилку.

Ця помилка пов’язана з тим, що в **PowerShell** параметри команди завжди повинні відділятися від імені самої команди проміжком. Тому останню команду потрібно виконувати таким чином:

***PS C:\Windows> cd ***

Отримання списку файлів і каталогів

Нагадаємо, що в оболонці **cmd.exe** список файлів і каталогів формується за допомогою внутрішньої команди ***dir***. У **PowerShell** також можна використовувати команду ***dir***, яка є псевдонімом командлета ***Get-ChildItem***. Інші псевдоніми – ***ls***, ***gci***.

У шляху, який вказується для команди ***dir***, можна застосовувати групові символи підстановки. Наприклад, наступна команда виведе всі файли з розширенням ***log*** з каталогу ***C:\Windows***:

PS C:\> dir C:\Windows*.log

Параметр ***-Exclude*** дозволяє задати маску файлів, які не будуть оброблюватись командою ***dir***. Наприклад, зазначена команда виведе

всі файли з розширенням ***log*** з каталогу ***C:\Windows***, крім тих, чиє ім'я починається на букву ***d***:

PS C:\> dir C:\Windows*.log -Exclude d*.log

Параметр ***-Name*** дозволяє виводити на екран тільки імена файлів (таким чином, цей параметр є аналогом ключа ***/b*** команди ***dir*** з **cmd.exe**), наприклад:

PS C:\> dir C:\Windows*.log –Name

Параметр ***-Recurse*** вмикає режим рекурсії, за якого командлет ***dir*** відображає не тільки вміст зазначеного каталогу, але і всіх його підкаталогів:

PS C:\> dir 'Documents and Settings' –Recurse

За замовчуванням командлет ***dir*** «не бачить» приховані файли. Якщо необхідно такі файли також включати в список, то потрібно вказати параметр ***-Force***:

PS C:\> dir C:\Windows –Force

Створення файлів і каталогів

Створити новий файл або каталог у **PowerShell** дозволяє командлет ***New-Item***. Шлях до створюваного елемента вказується у вигляді значення параметра ***-Path***, а як значення параметра ***-Type*** вказується "***directory***", якщо потрібно створити каталог, і "***file***", якщо потрібно створити файл. Наприклад, подана команда створює на диску ***C:*** каталог з ім'ям ***test folder***:

PS C:\> New-Item -Path C:\test_folder -Type "directory"

Під час створення файлу в нього відразу можна записати рядок, вказавши його як значення параметра ***-Value***, наприклад:

PS C:\> New-Item -Path C:\test_file.txt -Type "file" -Value "Test"

Якщо спробувати назвати створюваний файл ім'ям вже існуючого файлу ***PS C:\> New-Item -Path C:\test_file.txt -Type "file" -Value "Test"***, то виникне помилка.

Для перезапису існуючого файлу під час створення потрібно вказати параметр ***-Force***: ***PS C:\> New-Item -Path C:\test_file.txt -Type "file" -Value "Test2" –Force***.

Читання і перегляд вмісту файлів

В оболонці **cmd.exe** є команда ***type***, яка виводить вміст текстового файлу на екран. У **PowerShell** команда ***type*** є псевдонімом командлету ***Get-Content*** (інші псевдоніми цього ж командлету – ***cat*** і ***gc***), призначеного для порядкового зчитування вмісту текстового файлу з

поверненням об’єкта для кожного рядка (при цьому рядки відображаються на екрані). Наприклад:

PS C:\> Get-Content C:\Windows\win.ini

Параметр ***-Encoding*** командлет ***Get-Content*** дозволяє явно вказувати кодування файлу для коректної обробки його вмісту. Допустимі значення цього параметра: ***Unicode***, ***Byte***, ***BigEndianUnicode***, ***UTF8***, ***UTF7***, ***Ascii***.

За замовчуванням командлет ***Get-Content*** зчитує всі рядки з файлу; їх кількість можна обмежити за допомогою параметра ***-TotalCount***. Наприклад, наступна команда зчитує перші п’ять рядків з файлу ***C:\Windows\win.ini***:

PS C:\> Get-Content C:\Windows\win.ini -TotalCount 5

Запис файлів

Записати дані у зовнішні файли можна за допомогою операторів перенаправлення (> і >>) і командлет ***Out-File***. При цьому командлет ***Out-File*** намагається форматувати у файл об’єкти, що записуються. Якщо потрібно просто записати у файл текстову інформацію (без додаткового форматування), то краще скористатися командлетом ***Set-Content***.

Дані для запису у файл можуть задаватися як значення параметра ***-Value***. Наприклад, наступна команда записує у файл ***C:\test.txt*** рядок ***"Рядок з PowerShell"***:

PS C:\> Set-Content C:\test.txt -Value "Рядок з PowerShell"

Копіювання файлів і каталогів

У **PowerShell** копіювання файлів і каталогів здійснюється командлетом ***Copy-Item***, які мають псевдонім ***copy***. Шлях до файлів, що копіюються, при цьому вказується як значення параметра ***-Path*** (цей параметр використовується за замовчуванням), а шлях до цільового каталогу, в який потрібно скопірювати файли, задається значенням параметра ***-Destination***. Наприклад, ця команда скопіює файл ***styles.css*** з кореневого каталогу диска C: в каталог ***C:\test_folder***:

PS C:\> copy C:\Styles.css -Destination C:\test_folder

Для того, щоб побачити результат виконання команди копіювання, потрібно вказати параметр ***-PassThru***:

PS C:\> copy C:\styles.css -Destination C:\test_folder –PassThru

Якщо шлях до об’єктів, що копіюються, вказує на каталог, то за замовчуванням буде скопійований тільки цей каталог без свого вмісту

(цим **PowerShell** відрізняється від більшості інших оболонок, в тому числі від ***cmd.exe***). Наприклад:

PS C:\> copy C:\script -Destination C:\test_folder -PassThru

Параметр ***-Recurse*** дозволяє копіювати вміст вкладених каталогів, наприклад:

PS C:\> copy C:\script -Destination C:\test_folder -Recurse -PassThru

Можна копіювати не всі файли з каталогу, а тільки відповідні певній масці. При цьому маску можна вказати всередині шляху для копіювання або як значення параметра ***-Include***. Наприклад, вказана команда копіює всі файли з розширенням ***psl*** з каталогу ***C:\script*** в папку ***C:\test_folder***:

PS C:\> copy C:\script*.psl -Destination C:\test_folder –PassThru

Однак якщо необхідно скопіювати і підкаталоги, то одним командлетом ***Copy-Item*** обійтися не вдасться. Попередньо необхідні файли потрібно отримати командлетом ***Get-ChildItem (dir***), а потім передати їх командлету ***Copy-Item*** по конвеєру. Наприклад, наступна команда копіює всі файли з розширенням ***psl*** з каталогу ***C:\script*** і всіх його підкаталогів в папку ***C:\test_folder***:

PS C:\> dir -Recurse -Include *.psl c:\script* | copy -Destination C:\test_folder -PassThru

Команда ***copy*** оболонки **cmd.exe** дозволяла об'єднувати кілька файлів (конкатенація файлів). У **PowerShell** об'єднати файли можна за допомогою командлету ***Get-Content*** (псевдонім ***type***) і перенаправлення виведення в результуючий файл. Розглянемо приклад. Створимо файли ***l.txt*** і ***2.txt***:

PS C:\> New-Item -Path C:\l.txt -Type "file" -Value "File 1"
PS C:\> New-Item -Path C:\2.txt -Type "file" -Value "File 2"

Наступна команда об'єднує файли ***1.txt*** і ***2.txt*** у файл ***3.txt***:

PS C:\> type l.txt, 2.txt > .\3.txt

Перейменування і переміщення файлів і каталогів

Перейменувати файл або каталог можна за допомогою командлета ***Rename-Item*** (псевдонім ***ren***). Значення параметра ***-Path*** цього командлета задає шлях до елементів для перейменування, а значення параметра ***–NewName*** – нове ім'я. Імена цих параметрів можна опускати (в цьому випадку першим має зазначатися значення параметра ***-Path***). Наприклад, створимо файл ***c:\l.tmp*** і перейменуємо його у файл ***2.tmp***:

PS C:\> New-Item -Path C:\l.tmp -Type "file"
PS C:\> ren l.tmp 2.tmp

Для того, щоб побачити результат дії командлета ***Rename-Item***, потрібно вказати параметр ***-PassThru***:

PS C:\> ren 2.tmp 3.tmp –PassThru

Командлет ***Rename-Item*** дозволяє лише перейменовувати файли або каталоги, а не переміщувати їх. Якщо потрібно перемістити файл або каталог в іншу папку, то слід скористатися командлетом ***Move-item*** (псевдонім ***move***). Значення параметра ***-Path*** цього командлета задає шлях до файлів або каталогів для переміщення (в цьому шляху допускається використання символів узагальнення), а значення параметра ***-Destination*** – шлях до каталогу, куди будуть переміщені ці файли або каталоги. Результат перемещення можна побачити на екрані, вказавши параметр ***-PassThru***. Наприклад, така команда перенесе у кореневий каталог диску ***C:*** каталог ***C:\test_folder\folderl*** з усім його вмістом:

PS C:\> Move-Item -Path C:\test_folder\folderl C:\ -PassThru

Видаляти об'єкти файлової системи можна за допомогою командлета ***Remove-Item*** (псевдонім ***del***). Значення параметра ***-Path*** цього командлета задає шлях до файлів або каталогів, що видаляються (ім'я параметра в команді можна не вказувати). У шляху допускаються групові символи. Крім того, командлет ***Remove-Item*** має параметр ***-Include***, значення якого задає файли, на які діятиме команда, і параметр ***-Exclude***, що задає файли-виключення, які видалятися не будуть.

Наприклад, наступна команда видалить всі файли з розширенням ***psl*** у каталозі ***C:\test_folder***:

PS C:\> del C:\test_folder*.psl

Якщо спробувати видалити всі файли в каталозі, що має підкаталоги, то система видасть попередження:

PS C:\> del C:\test_folder*

3.3. Конвеєризація об'єктів у PowerShell

Раніше ми вже неодноразово казали про те, що всі дії в оболонці PowerShell пов'язані з операціями над об'єктами.

Незайвим буде нагадати, що об'єкт – це сукупність даних (властивості об'єкта) і способів роботи з цими даними (методи об'єкта). Конкретна структура об'єкта (складу властивостей і методів) задається його типом (наприклад, будь-якому файлу на твердому диску відповідає об'єкт типу Fileinfo). Набір типів, що використовуються в PowerShell, базується на типах уніфікованої платформи

.NET Framework, повсюдно використовується в сучасних версіях операційної системи Windows.

Властивість об’єкта – це відомості про стан або параметри. Наприклад, у об’єкта Fileinfo є властивість Length (довжина), що відповідає розміру файлу, який представлений таким об’єктом.

Метод об’єкта є дією, яку можна здійснювати над елементом, представленим таким об’єктом. Наприклад, у об’єкта Fileinfo є метод copyTo, за допомогою якого можна скопіювати файл (у разі виклику цього методу відбувається копіювання представленого об’єктом файлу на рівні файлової системи).

Властивості і методи об’єктів використовуються в командлетах PowerShell для виконання різних дій і роботи з даними. При цьому в PowerShell підтримується запозичений з інших інтерфейсів командного рядка механізм конвеєризації або композиції команд, що значно підвищує ефективність роботи.

У більшості оболонок командного рядка, включаючи ***cmd.exe***, під конвеєризацією розуміється об’єднання (композиція) кількох команд шляхом послідовного перенаправлення вихідного потоку однієї команди у вхідний потік іншої, що дозволяє передавати текстову інформацію між різними процесами.

Механізм композиції команд має, ймовірно, найбільш цінну концепцію, яка використовується в інтерфейсах командного рядка. Конвеєри не тільки знижують зусилля, прикладені під час введення складних команд, але і полегшують відстеження виконуваних командами дій. Корисною рисою конвеєрів є те, що вони не залежать від числа переданих елементів, оскільки конвеєр діє на кожен елемент окремо. Крім того, кожна команда в конвеєрі (звана елементом конвеєра) зазвичай передає свій висновок наступній команді в конвеєрі, елемент за елементом. Завдяки цьому, як правило, знижується споживання ресурсів для складних команд і з’являється можливість отримувати інформацію, що виводиться негайно.

В оболонці **PowerShell** також дуже широко використовується механізм конвеєризації команд, проте тут по конвеєру передається не потік тексту, як у всіх інших оболонках, а **об’єкти**. При цьому з елементами конвеєра можна виробляти різні маніпуляції: фільтрувати об’єкти але певним критерієм, сортувати і групувати об’єкти, змінювати їх структуру.

Конвеєр в PowerShell – це послідовність команд, розділених між собою знаком **/** (вертикальна риска). Кожна команда в конвеєрі отримує об’єкт від попередньої команди, виконує певні операції над ним та передає наступній команді в конвеєрі. З точки зору користувача, об’єкти упаковують пов’язану інформацію у форму, в

якій інформацією простіше маніпулювати як єдиним блоком і з якої за необхідності витягуються певні елементи.

Передача даних між командами у вигляді об'єктів має велику перевагу над звичайним обміном інформацією за допомогою потоку тексту. Адже команда, яка бере потік тексту від іншої утиліти, повинна його проаналізувати, розібрати і виділити потрібну їй інформацію, а це може бути непросто, оскільки зазвичай виведення команди більше орієнтоване на візуальне сприйняття людиною (це природно для інтерактивного режиму роботи), а не на зручність подальшого синтаксичного розбору.

Під час передачі по конвеєру об'єктів цієї проблеми не виникає, тут потрібна інформація витягується з елемента конвеєра простим зверненням до відповідної властивості об'єкта. Однак тепер виникає нове запитання: як можна дізнатися, які саме властивості є у об'єктів, що передаються по конвеєру? Адже у разі виконання того чи іншого командлета ми на екрані бачимо тільки одну або кілька колонок відформатованого тексту. Наприклад, запустимо командлет ***Get-Process***, який виводить інформацію про запущені в системі процеси:

```
PS F:\> Get-Process

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    129       7     1468       1760              8032   0 AppVShNotify
    114       8     1984       1648              2896   0 armsvc
    211      11     2192      10096              8880  14 atieclxx
    123       7     1252       2320              1660   0 atiesrxx
    191      13     6752      13600       0,36   1944   0 audiodg
   3274      10     5860       1684       0,42   3840  14 CAudioFilterAgent64
    156      12     2556       9188       5,63   5532  14 CNMNSST2
    114      10     2852       9556      29,52  10580  14 conhost
    449      14     1564       1804               676   0 csrss
    522      15     2024       7280              1316  14 csrss
    155       9     1704       1636              2904   0 CxAudMsg64
    153      10     1472       2976              2912   0 CxUtilSvc
     67       5      868        760              8688   0 dasHost
    130       8     1732       9184       0,17   6720  14 dllhost
    508      32    44096      47152              6752  14 dwm
   2474     180    65500     130556     160,14   6280  14 explorer
     37       7      996       5564              7284  14 fontdrvhost
    154      10     1708        700              6516   0 GoogleCrashHandler
    137       8     1544        204              5224   0 GoogleCrashHandler64
    849      47   229384     248912     129,55   9312  14 googledrivesync
     36       5     3024       3568       6,22  12076  14 googledrivesync
      0       0        0          4                 0   0 Idle
    125       8     1292       1824              2072   0 ijplmsvc
   1279      23     7468      12452               956   0 lsass
      0       0      240      17852              1928   0 Memory Compression
```

Фактично на екрані ми бачимо тільки зведену інформацію (результат форматування отриманих даних), а не повне представлення вихідного об'єкта. З цієї інформації незрозуміло, скільки точно властивостей є у об'єктів, що генеруються командою ***Get-Process***, і які імена мають ці властивості. Наприклад, ми хочемо знайти всі «зависли» процеси, які не відповідають на запити системи. Чи можна

це зробити за допомогою командлета ***Get-Process***, яку саме властивість для цього потрібно перевіряти у виведених об’єктів?

Для відповіді на ці запитання потрібно, перш за все, навчитися досліджувати структуру об’єктів **PowerShell**, дізнаватися, які властивості і методи є у цих об’єктів.

3.3.1. Командлет Get-Member – перегляд структури об’єктів

Для аналізу структури об’єкта, що повертається певною командою, простіше направити цей об’єкт по конвеєру на командлет ***Get-Member*** (псевдонім ***gm***), наприклад:

```
PS F:\> Get-Process | Get-Member

   TypeName: System.Diagnostics.Process

Name                       MemberType     Definition
----                       ----------     ----------
Handles                    AliasProperty  Handles = Handlecount
Name                       AliasProperty  Name = ProcessName
NPM                        AliasProperty  NPM = NonpagedSystemMemorySize64
PM                         AliasProperty  PM = PagedMemorySize64
SI                         AliasProperty  SI = SessionId
VM                         AliasProperty  VM = VirtualMemorySize64
WS                         AliasProperty  WS = WorkingSet64
Disposed                   Event          System.EventHandler Disposed(System.Object, System.EventArgs)
ErrorDataReceived          Event          System.Diagnostics.DataReceivedEventHandler ErrorDataReceived(System.Objec...
Exited                     Event          System.EventHandler Exited(System.Object, System.EventArgs)
OutputDataReceived         Event          System.Diagnostics.DataReceivedEventHandler OutputDataReceived(System.Obje...
BeginErrorReadLine         Method         void BeginErrorReadLine()
BeginOutputReadLine        Method         void BeginOutputReadLine()
CancelErrorRead            Method         void CancelErrorRead()
CancelOutputRead           Method         void CancelOutputRead()
Close                      Method         void Close()
CloseMainWindow            Method         bool CloseMainWindow()
```

У результаті на екрані ми бачимо, який .NET-тип мають об’єкти, які повертаються в ході роботи досліджуваного командлет (у нашому прикладі це тип ***System.Diagnostic.Process***), а також повний список елементів об’єкта. При цьому на екран виводиться дуже багато елементів різних типів (імена і псевдонім властивостей, імена методів і т. д.), і такий довгий список стає незручно переглядати. Командлет ***Get-Member*** має параметр ***-MemberType***, що дозволяє перерахувати тільки елементи об’єкта певного типу. Наприклад, для виведення тільки елементів об’єкта, що є властивостями цього об’єкта, використовується параметр ***-MemberType*** зі значенням ***Property***:

```
PS F:\> Get-Process | Get-Member -MemberType Property

   TypeName: System.Diagnostics.Process

Name                       MemberType Definition
----                       ---------- ----------
BasePriority               Property   int BasePriority {get;}
Container                  Property   System.ComponentModel.IContainer Container {get;}
EnableRaisingEvents        Property   bool EnableRaisingEvents {get;set;}
ExitCode                   Property   int ExitCode {get;}
ExitTime                   Property   datetime ExitTime {get;}
Handle                     Property   System.IntPtr Handle {get;}
HandleCount                Property   int HandleCount {get;}
HasExited                  Property   bool HasExited {get;}
Id                         Property   int Id {get;}
MachineName                Property   string MachineName {get;}
MainModule                 Property   System.Diagnostics.ProcessModule MainModule {get;}
MainWindowHandle           Property   System.IntPtr MainWindowHandle {get;}
MainWindowTitle            Property   string MainWindowTitle {get;}
MaxWorkingSet              Property   System.IntPtr MaxWorkingSet {get;set;}
MinWorkingSet              Property   System.IntPtr MinWorkingSet {get;set;}
```

Як бачите, процесам операційної системи відповідають об'єкти, що мають дуже багато властивостей, на екран же під час роботи командлета ***Get-Process*** виводяться лише кілька з них. Насправді способи відображення в оболонці **PowerShell** об'єктів різних типів задаються декількома файлами з розширенням **pslxml** у форматі XML, які знаходяться в каталозі, де встановлений файл **powershell.exe** (шлях до цього каталогу зберігається в змінній ***$PSHome***). Список цих файлів можна отримати за допомогою такої команди:

PS C:\> dir $pshome*format*.pslxml

Зокрема, правило форматування об'єкта типу ***System.Diagnostic. Process*** знаходиться у файлі **dotnettypes.format.pslxml**. Редагувати безпосередньо конфігураційні файли не рекомендується, а в разі необхідності можна створити власні файли форматування і за допомогою командлета ***Update-FormatData*** включити їх до складу файлів, що автоматично завантажуються.

Тепер, коли ми знаємо, які властивості мають об'єкти, що передаються по конвеєру, перейдемо до розгляду можливих операцій над елементами конвеєра.

3.3.2. Командлет Where-Object – фільтрація об'єктів

У **PowerShell** підтримується можливість фільтрації об'єктів у конвеєрі, тобто видалення з конвеєра об'єктів, які задовольняють певні умови. Таку функціональність забезпечує командлет ***Where-Object***, що дозволяє перевірити кожний об'єкт, що проходить через конвеєр, і передати його далі по конвеєру лише в тому випадку, якщо об'єкт задовольняє умову перевірки.

Умова перевірки у ***Where-Object*** задається у вигляді блоку сценарію (**scriptblock**) – однієї або декількох команд **PowerShell,** укладених у фігурні дужки ***{}***. Блок сценарію вказується після імені командлета ***Where-Object***. Результатом виконання блоку сценарію в командлеті ***Where-Object*** має бути значення логічного типу: ***$True*** (істина, в цьому випадку об'єкт проходить далі по конвеєру) або ***$False*** (брехня, в цьому випадку об'єкт далі по конвеєру не передається).

Наприклад, для виведення інформації про зупинені служби в системі (об'єкти, які повертаються командлетом ***Get-Service***, у яких властивість ***Status*** дорівнює "***stopped***") можна використовувати такий конвеєр:

```
PS F:\> Get-Service | Where-Object {$_.Status -eq "Stopped"}

Status   Name               DisplayName
------   ----               -----------
Stopped  AJRouter           AllJoyn Router Service
Stopped  ALG                Application Layer Gateway Service
Stopped  AppIDSvc           Application Identity
Stopped  AppReadiness       App Readiness
Stopped  AppVClient         Microsoft App-V Client
Stopped  AppXSvc            AppX Deployment Service (AppXSVC)
Stopped  aspnet_state       ASP.NET State Service
Stopped  AvastWscReporter   AvastWscReporter
Stopped  AxInstSV           ActiveX Installer (AxInstSV)
Stopped  BITS               Background Intelligent Transfer Ser...
Stopped  bthserv            Bluetooth Support Service
Stopped  ClipSVC            Client License Service (ClipSVC)
Stopped  COMSysApp          COM+ System Application
Stopped  CscService         Offline Files
Stopped  DcpSvc             DataCollectionPublishingService
Stopped  defragsvc          Optimize drives
Stopped  DeviceAssociati... Device Association Service
Stopped  DeviceInstall      Device Install Service
```

Інший приклад – залишимо в конвеєрі тільки ті процеси, у яких значення ідентифікатора (властивість ***id***) більше 4000:

```
PS F:\> Get-Process | Where-Object {$_.Id -gt 4000}

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    274      21    19472      21388       1,19   6192  14 AnyDesk
    241      17    20492      23236       0,36   9712  14 AnyDesk
    129       7     1468       1760              8032   0 AppVShNotify
    211      11     2192      10124              8880  14 atieclxx
    294      16    40724      46440       1,36   8948   0 audiodg
    154      12     2548       9188       6,17   5532  14 CNMNSST2
    174      12     4320      14020       0,23  12092  14 conhost
     67       5      868        760              8688   0 dasHost
    161      16     3308       9952       0,20   6720  14 dllhost
    479      33    53412      47176              6752  14 dwm
   2573     182    73844     140712     171,50   6280  14 explorer
     37       7      992       5560              7284  14 fontdrvhost
```

У блоках сценаріїв командлет ***Where-Object*** для звернення до поточного об’єкта конвеєра і вилучення потрібних властивостей цього об’єкта використовується спеціальна змінна ***$_***, яка створюється оболонкою **PowerShell** автоматично. Ця змінна використовується і в інших командлетах, які виробляють обробку елементів конвеєра.

Як можна зрозуміти з прикладів, у блоці сценарію використовуються спеціальні оператори порівняння. Основні оператори порівняння наведені в табл. 3.1.

У **PowerShell** для операторів порівняння не використовуються звичайні символи «>» або «<», оскільки в командному рядку вони зазвичай означають перенаправлення введення / виведення.

Табл. 3.1

Оператори порівняння в **PowerShell**

Оператор	Значення	Приклад (повертається значення *True*)
-eq	дорівнює	***10 -eq 10***
-ne	не дорівнює	***9 -ne 10***
-lt	менше	***3 -lt 4***
-le	менше або дорівнює	***3 -le 4***
-gt	більше	***4 -gt 3***
-ge	більше або дорівнює	***4 -ge 3***
-like	порівняння на збіг з урахуванням символів узагальнення у другому операнді	***"file.doc" -like "f *.doc"***
-notlike	порівняння на розбіжність з урахуванням символів узагальнення в другому операнді	***"fiie.doc" -notlike "f*.rtf"***
-contains	містить	***1,2,3 -contains 1***
-notcontains	не містить	***1,2,3 -notcontains 4***

Оператори порівняння можна з'єднувати один з одним за допомогою логічних операторів (табл. 3.2).

Табл. 3.2

Логічні оператори в **PowerShell**

Оператор	Значення	Приклад (повертається значення *True*)
-and	логічне І	***(10 -eq 10) -and (1 -eq 1)***
-or	логічне АБО	***(9 -ne 10) -or (3 -eq 4)***
-not	логічне НЕ	***-not (3 -gt 4)***
!	логічне НЕ	***!(3 -gt 4)***

3.3.3. Командлет Sort-Object – сортування об'єктів

Сортування елементів конвеєра – ще одна операція, яка часто застосовується під час конвеєрної обробки об'єктів. Цю операцію здійснює командлет ***Sort-Object***: йому передаються імена властивостей, за якими потрібно провести сортування об'єктів, що проходять по конвеєру, а він повертає дані, впорядковані за значеннями цих властивостей.

Наприклад, для виведення списку запущених у системі процесів, впорядкованого за їх ідентифікаторами (властивість ***Id***), можна скористатись таким конвеєром:

```
PS F:\> Get-Process | Sort-Object -Property Id

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
      0       0        0          4                 0   0 Idle
   1574       0      160      13060                 4   0 System
    766      26     7908      13912                 8   0 svchost
     51       3      372        556               452   0 smss
    454      15     1624       1844               676   0 csrss
    669      19     6304       8584               708   0 svchost
    117       9     1400       3168               812   0 wininit
    295      10     3592       5080               948   0 services
   1301      23     7808      12576               956   0 lsass
    112       9     2228       2416               992   0 sqlwriter
   2335     121    39684      48316              1096   0 svchost
    864      42    12304      19632              1104   0 svchost
    134       9     1292       6904       0,05   1200  14 schedhlp
    711      25    14920      16688              1204   0 svchost
```

Параметр ***-Property*** в командлеті ***Sort-Object*** використовується за замовчуванням, тому ім’я цього параметра можна не вказувати. Для сортування в зворотному порядку використовується параметр ***-Descending***.

```
PS F:\> Get-Process | Sort-Object Id -Descending

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    199      14     2532      11004       0,14  12132  14 svchost
    178      12     4240      12660       1,63  12092  14 conhost
     36       5     3024       3568       6,22  12076  14 googledrivesync
    919      43    27880      65936       8,30  11376  14 ShellExperienceHost
    372      22    10420      33972       9,89  11356  14 RuntimeBroker
    174      14     8472      15076       0,08  10868  14 smartscreen
    710      28    70036      76904       2,02  10652  14 powershell
   1266      42    64028       5108       2,98  10272  14 SmartAudio
    499      34   161284     177672      48,11  10196  14 Telegram
    242      17    20492      22396       0,36   9712  14 AnyDesk
    844      47   229316     248864     134,28   9312  14 googledrivesync
    674      69    34720      29252              9304   0 SearchIndexer
    713      26    45112      54112              9144   0 OfficeClickToRun
    388      15     5004      21080       1,22   9100  14 sihost
    290      16    40696      15556       2,86   8948   0 audiodg
    207      11     2128      10132              8880  14 atieclxx
```

У розглянутих нами прикладах конвеєри складалися з двох командлетів. Це не обов’язкова умова, конвеєр може об’єднувати і більшу кількість команд, наприклад:

```
PS F:\> Get-Process | Where-Object {$_.Id -gt 1000} | Sort-Object Id -Descending

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    653      33    22344      57260       1,25  12152  15 ShellExperienceHost
    121       8     5988       7104       0,11  11952  15 chrome
    168      17     4080      10320       0,09  11928  15 dllhost
    530      16     2236       7096             11164  15 csrss
   1164      44    80340     145708      39,02  10564  15 chrome
    188      14    17508      22580       0,14   9640  15 chrome
    136      10     1496       7368       0,03   9512  15 schedhlp
    143      10     2532       8700              9424   0 WmiPrvSE
    669      69    34468      29860              9304   0 SearchIndexer
    115      10     3008       9488       0,28   9192  15 conhost
    712      26    45252      54504              9144   0 OfficeClickToRun
    464      32   146740     161760      11,23   9012  15 Telegram
     67       5      868        760              8688   0 dasHost
    236      15    13136      18092       0,23   8624  15 chrome
    191      13     7212      13916       0,69   8576   0 audiodg
   1866     102    44548     104412      28,52   8240  15 explorer
    129       7     1500       1772              8032   0 AppVShNotify
```

3.3.4. Командлет Select-Object – виділення об’єктів та властивостей

У **PowerShell** є командлет ***Select-Object***, за допомогою якого можна виділяти вказану кількість об’єктів з початку або з кінця конвеєра, вибирати унікальні об’єкти з конвеєра, а також виділяти певні властивості в об’єктах, що проходять по конвеєру.

Для виділення з конвеєра декількох перших або останніх об’єктів слід скористатися відповідно параметрами ***-First*** або ***-Last*** командлета ***Select-Object***. Наприклад, вказаний конвеєр команд виведе на екран інформацію про п’ять процесів, що використовують найбільший обсяг пам’яті:

```
PS F:\> Get-Process | Sort-Object WS | Select-Object -Last 5

Handles  NPM(K)    PM(K)      WS(K)     CPU(s)     Id  SI ProcessName
-------  ------    -----      -----     ------     --  -- -----------
    601      26   179648     163364      43,72   6348  15 chrome
   1404      67   130460     170004      70,33   2524  15 WINWORD
   6968     100   170316     206988      14,59   3512  15 Viber
    800      47   232792     253860      87,78   4168  15 googledrivesync
   1018      92   387988     305556              3240   0 MsMpEng
```

Розберемо роботу цього конвеєра команд. Перший командлет у конвеєрі (***Get-Process***) повертає масив об’єктів, відповідних запущеним у системі процесам. Другий командлет ***Sort-Object*** впорядковує об’єкти, проходять по конвеєру, за значенням властивості ***WS*** (обсяг пам’яті, займаної процесом). Нарешті, третій командлет ***Select-Object*** вибирає з упорядкованого масиву об’єкта останні п’ять елементів.

Припустимо тепер, що нам потрібно отримати список запущених у системі процесів, в якому були б вказані тільки імена процесів і їх ідентифікатори. Якщо ви не пам’ятаєте назви потрібних властивостей, то можна за допомогою командлета ***Get-Member*** знову переглянути структуру об’єктів, що повертаються командою ***Get-Process***:

```
PS F:\> Get-Process | Get-Member -MemberType Property


   TypeName: System.Diagnostics.Process

Name                       MemberType Definition
----                       ---------- ----------
BasePriority               Property   int BasePriority {get;}
Container                  Property   System.ComponentModel.IContainer Container {get;}
EnableRaisingEvents        Property   bool EnableRaisingEvents {get;set;}
ExitCode                   Property   int ExitCode {get;}
ExitTime                   Property   datetime ExitTime {get;}
Handle                     Property   System.IntPtr Handle {get;}
HandleCount                Property   int HandleCount {get;}
HasExited                  Property   bool HasExited {get;}
Id                         Property   int Id {get;}
MachineName                Property   string MachineName {get;}
MainModule                 Property   System.Diagnostics.ProcessModule MainModule {get;}
```

Отже, в підсумкових об’єктах нам потрібно залишити тільки властивості ***ProcessName*** і ***Id***. Це можна зробити, вказавши імена потрібних властивостей як параметрів командлет ***Select-Object***:

```
PS F:\> Get-Process | Select-Object ProcessName, Id

ProcessName          Id
-----------          --
AppVShNotify       8032
armsvc             2896
atieclxx            976
atiesrxx           1660
audiodg           10148
CAudioFilterAgent64  5616
CNMNSST2           2796
conhost            9192
csrss               676
csrss             11164
CxAudMsg64         2904
CxUtilSvc          2912
dasHost            8688
```

Подивимося тепер, який тип має об’єкт, що формується в конвеєрі командлетом ***Select-Object***, і які властивості є у цього об’єкта:

```
PS F:\> Get-Process | Select-Object ProcessName, Id | Get-Member


   TypeName: Selected.System.Diagnostics.Process

Name        MemberType   Definition
----        ----------   ----------
Equals      Method       bool Equals(System.Object obj)
GetHashCode Method       int GetHashCode()
GetType     Method       type GetType()
ToString    Method       string ToString()
Id          NoteProperty int Id=8032
ProcessName NoteProperty string ProcessName=AppVShNotify
```

Як бачите, вихідний об’єкт має тип ***System.Management. Automation. PSCustomObject*** (нагадаємо, що командлет ***Get-Process*** повертав об’єкти типу ***System.Diagnostics.Process***). І у нього є тільки дві властивості ***ProcessName*** і ***Id***. Це пов’язано з тим, що під час використання командлету ***Select-Object*** для вибору зазначених власти-востей він копіює значення цих властивостей з об’єктів, що надходять по конвеєру йому на вхід, і створює нові об’єкти, які містять зазначені властивості зі скопійованими значеннями.

Командлет ***Select-Object*** може не тільки видаляти з об’єктів непотрібні властивості, але і додавати нові обчислювані властивості. Для цього нову властивість потрібно представити у вигляді хеш-

таблиці, де перший елемент (ключ ***Name***) відповідає імені властивості, яка додається, а другий елемент (ключ ***Expression***) – значенням цієї властивості для поточного елемента конвеєра.

Наприклад, результатом виконання наступного конвеєра команд стане масив об’єктів, що мають властивості ***ProcessName*** (ім’я запущеного процесу) і ***startMin*** (хвилина, коли був запущений процес):

```
PS F:\> Get-Process | Select-Object ProcessName, @{Name="StartMin"; Expression={$_.StartTime.Minute}}

ProcessName          StartMin
-----------          --------
AppVShNotify
armsvc
atieclxx
atiesrxx
audiodg              9
CAudioFilterAgent64  39
CNMNSST2             38
conhost              53
csrss
csrss
CxAudMsg64
CxUtilSvc
dasHost
dllhost              53
dwm
explorer             38
```

Тут властивість ***startMin*** є обчислюваним, його значення для кожного елемента конвеєра задається блоком коду ***{$_.StartTime .Minute}***, де змінна відповідає поточному об’єкту конвеєра.

3.3.5. Командлет ForEach-Object – виконання довільних дій над об’єктами у конвеєрі

Командлет ***ForEach-Object*** дозволяє виконати певний блок сценарію (код мовою **PowerShell**) для кожного об’єкта в конвеєрі. Іншими словами, за допомогою цього командлета можна виробляти довільні операції над елементами конвеєра. Для прикладу давайте підрахуємо загальний обсяг файлів, що зберігаються у певному каталозі диска. Для цього оголосимо змінну ***$TotalLength*** і присвоїмо її значення нулю:

PS D:\Gleb> $TotalLength=0

Тепер виконаємо команду ***dir*** і результат її роботи передамо по конвеєру командлета ***ForEach-Object***:

PS D:\Gleb> dir | ForEach-Object {$TotalLength+=$_.Length}

У блоці сценарію командлет ***ForEach-Object*** до поточного значення змінної ***$TotalLength*** додається значення властивості ***Length*** проходити через конвеєр об’єкта (розмір відповідного цьому об’єкту файлу). У результаті у змінній ***$TotalLength*** буде зберігатися загальний розмір файлів у байтах:

PS D:\Gleb> $TotalLength
20336777

3.3.6. Командлет Group-Object – групування об’єктів

Об’єкти, що проходять по конвеєру, можна згрупувати за значенням певних властивостей за допомогою командлета ***Group-Object***. В одну групу будуть потрапляти об’єкти, що мають однакові значення зазначених властивостей (властивості можуть бути обчислювані).

Розглянемо приклад. Командлет ***Get-Process*** генерує об’єкти, що мають властивості ***Company*** (назва компанії-розробника певного модуля, запущеного в операційній системі як процес). Виконаємо групування цих об’єктів за значенням властивості ***Company***:

```
PS F:\> Get-Process | Group-Object Company

Count Name                      Group
----- ----                      -----
   49                           {System.Diagnostics.Process (AppVShNotify), System.Diagnostics.Process (armsvc), Sys...
    1 Conexant Systems, Inc.    {System.Diagnostics.Process (CAudioFilterAgent64)}
    1 CANON INC.                {System.Diagnostics.Process (CNMNSST2)}
   14 Microsoft Corporation     {System.Diagnostics.Process (conhost), System.Diagnostics.Process (dllhost), System....
    2                           {System.Diagnostics.Process (googledrivesync), System.Diagnostics.Process (googledri...
    1 Acronis                   {System.Diagnostics.Process (schedhlp)}
    1 Conexant Systems, Inc     {System.Diagnostics.Process (SmartAudio)}
    1 McAfee, LLC               {System.Diagnostics.Process (SSScheduler)}
    1 Telegram FZ-LLC           {System.Diagnostics.Process (Telegram)}
    1 Viber Media S.Ã  r.l.     {System.Diagnostics.Process (Viber)}
```

Як бачите, в колонці ***Count*** відображається кількість елементів у кожній з груп, а в колонці ***Group*** перераховані елементи, що входять до групи.

Якщо потрібно просто дізнатися кількість елементів у групах, можна запустити командлет ***Group-Object*** з параметром ***-NoElement***:

```
PS F:\> Get-Process | Group-Object Company -NoElement

Count Name
----- ----
   49
    1 Conexant Systems, Inc.
    1 CANON INC.
   14 Microsoft Corporation
    2
    1 Acronis
    1 Conexant Systems, Inc
    1 McAfee, LLC
    1 Telegram FZ-LLC
    1 Viber Media S.Ã  r.l.
```

3.3.7. Командлет Measure-Object – вимірювання характеристик об’єктів

У **PowerShell** є ще один корисний командлет ***Measure-Object***, призначений для виконання функцій агрегування (сума, вибір мінімального, максимального або середнього значення) над властивостями елементів у конвеєрі об’єктів.

Розглянемо приклад. Раніше ми вже знаходили спільний розмір файлів у певному каталозі, застосовуючи для цього командлет ***ForEach-Object***.

За допомогою командлета ***Measure-Object*** ми також зможемо знайти сумарний розмір файлів. Для цього потрібно вказати, що ***Measure-Object*** повинен для всіх елементів конвеєра підсумувати (параметр ***-Sum***) значення властивості ***Length***:

```
PS F:\> dir | Measure-Object -Property Length -Sum


Count    : 15
Average  :
Sum      : 14085712083
Maximum  :
Minimum  :
Property : Length
```

Результат буде виведений у поле ***Sum***. Для виконання інших операцій потрібно вказати відповідний параметр: ***-Average*** для знаходження середнього значення, ***-Minimum*** або ***-Maximum*** для знаходження мінімального або максимального значення відповідно:

```
PS F:\> dir | Measure-Object -Property Length -Minimum -Maximum -Average -Sum


Count    : 15
Average  : 939047472,2
Sum      : 14085712083
Maximum  : 14040342554
Minimum  : 19
Property : Length
```

Також за допомогою командлета ***Measure-Object*** можна отримувати статистичну інформацію про текстові файли: кількість рядків, слів та символів.

3.4. Управління виведенням команд у PowerShell

У **PowerShell** є база даних (набір XML-файлів), що містить модулі форматування за замовчуванням для різних типів .NET-об’єктів. Ці модулі визначають, які властивості об’єкта відображуються під час виведення і в якому форматі: списку або таблиці. Нагадаємо, що командлети **PowerShell** повертають .NET-об’єкти, які, як правило, не знають, яким чином відображати себе на екрані. Коли об’єкт досягає кінця конвеєра, **PowerShell** визначає його тип і шукає його в списку об’єктів, для яких визначено правило форматування. Якщо такий тип у

списку виявлений, то до об’єкта застосовується відповідний модуль форматування; якщо немає, то **PowerShell** просто відображає властивості цього .NET-об’єкта.

Також у **PowerShell** можна явно задавати правила форматування даних, що виводяться командлетами, і як і в командному інтерпретаторі **cmd.exe** перенаправляти ці дані у файл, на принтер або у порожній пристрій.

3.4.1. Форматування виведеної інформації

У традиційних оболонках команди і утиліти самі форматують виведені дані. Деякі команди (наприклад, ***dir*** в інтерпретаторі **cmd.exe**) дозволяють налаштовувати формат виведення за допомогою спеціальних параметрів.

В оболонці **PowerShell** виведення форматують тільки чотири спеціальні командлети ***Format*** (табл. 3.3). Це спрощує вивчення, оскільки не потрібно запам’ятовувати засоби і параметри форматування для інших команд (інші командлети виведення не форматують).

Табл. 3.3

Командлети **PowerShell** для форматування виведення

Командлет	Опис
Format-Table	Форматує виведення у вигляді таблиці, стовпці якої містять властивості об’єкта (також можуть бути додані обчислювані стовпці). Підтримується можливість угруповання даних, що виводяться
Format-List	Виводить об’єкт як список властивостей. При цьому кожна властивість відображається на новому рядку. Підтримується можливість угруповання даних, що виводяться
Format-Custom	Використовує для користувача представлення (view) для форматування виведення
Format-Wide	Форматує об’єкти у вигляді широкої таблиці, в якій відображується тільки одна властивість кожного об’єкта

Якщо жоден з командлетів ***Format*** явно не вказаний, то використовується модуль форматування за замовчуванням, який визначається за типом даних, що відображаються. Наприклад, під час виконання командлету ***Get-Service*** дані за замовчуванням виводяться як таблиця з трьома стовпцями (***Status***, ***Name*** і ***DisplayName***):

```
PS F:\> Get-Service

Status   Name               DisplayName
------   ----               -----------
Stopped  AJRouter           AllJoyn Router Service
Stopped  ALG                Application Layer Gateway Service
Running  AnyDesk            AnyDesk Service
Running  Apache2.4          Apache2.4
Running  AppHostSvc         Application Host Helper Service
Stopped  AppIDSvc           Application Identity
Running  Appinfo            Application Information
Running  AppMgmt            Application Management
Stopped  AppReadiness       App Readiness
Stopped  AppVClient         Microsoft App-V Client
Stopped  AppXSvc            AppX Deployment Service (AppXSVC)
Stopped  aspnet_state       ASP.NET State Service
Running  AudioEndpointBu... Windows Audio Endpoint Builder
Running  Audiosrv           Windows Audio
```

Для зміни формату даних, що виводяться, потрібно направити їх по конвеєру відповідному командлету ***Format***. Наприклад, вказана команда виведе список служб за допомогою командлета ***Format-List***:

```
PS F:\> Get-Service | Format-List


Name                : AJRouter
DisplayName         : AllJoyn Router Service
Status              : Stopped
DependentServices   : {}
ServicesDependedOn  : {}
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : False
ServiceType         : Win32ShareProcess

Name                : ALG
DisplayName         : Application Layer Gateway Service
Status              : Stopped
DependentServices   : {}
ServicesDependedOn  : {}
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : False
ServiceType         : Win32OwnProcess
```

Як бачите, у результаті використання формату списку виводиться більше відомостей про кожну службу, ніж у форматі таблиці (замість трьох стовпців даних про кожну службу у форматі списку виводяться дев’ять рядків даних). Однак це зовсім не означає, що командлет ***Format-List*** витягує додаткові відомості про служби. Ці дані містяться в об’єктах, що повертаються командою ***Get-Service***, проте командлет, що використовується за замовчуванням, ***Format-Table*** відкидає їх, тому що не може вивести на екран більше трьох стовпців.

Під час форматування виведення за допомогою командлетів ***Format-List*** і ***Format-Table*** можна вказувати імена властивостей, які повинні бути відображені (нагадаємо, що переглянути список властивостей, наявних у об’єкта, дозволяє командлет ***Get-Member***).

Наприклад:

```
PS F:\> Get-Service | Format-List Name, Status, CanStop


Name    : AJRouter
Status  : Stopped
CanStop : False

Name    : ALG
Status  : Stopped
CanStop : False

Name    : AnyDesk
Status  : Running
CanStop : True
```

Вивести всі наявні у об’єктів властивості можна за допомогою параметра *. Наприклад:

```
PS F:\> Get-Service | Format-List *


Name                : AJRouter
RequiredServices    : {}
CanPauseAndContinue : False
CanShutdown         : False
CanStop             : False
DisplayName         : AllJoyn Router Service
DependentServices   : {}
MachineName         : .
ServiceName         : AJRouter
ServicesDependedOn  : {}
ServiceHandle       :
Status              : Stopped
ServiceType         : Win32ShareProcess
StartType           : Manual
Site                :
Container           :
```

3.4.2. Перенаправлення інформації, що виводиться

В оболонці **PowerShell** є кілька командлетів, за допомогою яких можна управляти виведенням даних. Ці командлети починаються зі слова ***Out***. Їх список можна побачити в такий спосіб:

```
PS F:\> Get-Command Out-* | Format-Table Name

Name
----
Out-Default
Out-File
Out-GridView
Out-Host
Out-Null
Out-Printer
Out-String
```

За замовчуванням виводиться інформація, передається командлету ***Out-Default***, який, у свою чергу, делегує всю роботу з виведення рядків на екран командлету ***Out-Host***. Для розуміння цього механізму потрібно враховувати, що архітектура **PowerShell** має на увазі відмінність між власне ядром оболонки (інтерпретатором команд) і головним додатком (host), який використовує це ядро. В принципі, як головний може виступати будь-який додаток, в якому реалізований ряд спеціальних інтерфейсів, що дозволяють коректно інтерпретувати отримувану від **PowerShell** інформацію. У нашому випадку головним додатком є консольне вікно, в якому ми працюємо з оболонкою, і командлет ***Out-Host*** передає інформацію, що виводиться в це консольне вікно.

Параметр ***-Paging*** командлета ***Out-Host***, подібно команді ***more*** інтерпретатора **cmd.exe**, дозволяє організувати посторінковий вивід інформації, наприклад:

PS F:\> Get-Help Get-Process -Full | Out-Host –Paging

3.4.3. Збереження даних у файл

Як уже згадувалося раніше, **PowerShell** підтримує перенаправлення виведення команд у текстові файли за допомогою стандартних операторів > і >>. Наприклад, наступна команда виведе вміст кореневого каталогу ***C:*** у текстовий файл ***D:\dir_c.txt*** (якщо такий файл існував, то він буде перезаписаний):

PS F:\> dir C:\ > dir_c.txt

Якщо потрібно перенаправити виведення команди у файл у режимі додавання (зі збереженням попереднього вмісту цього файлу), слід скористатися оператором >>:

PS F:\> dir C:\ >> dir_c.txt

Крім операторів перенаправлення > і >>, у **PowerShell** є командлет ***Out-File***, що також дозволяє направити виведені дані замість вікна консолі в текстовий файл. При цьому командлет ***Out-File*** має кілька додаткових параметрів, за допомогою яких можна більш гнучко керувати виведенням: задавати тип кодування файлу, задавати довжину виведених рядків у знаках, вибирати режим перезапису файлу (табл. 3.4).

Табл. 3.4

Деякі параметри командлета ***Out-File***

Параметр	Опис
-FilePath	Вказує шлях до вихідного файлу
-Encoding	Визначає кодування вихідного файлу. Можна вибрати зі значень **Unicode**, **UTF7**, **UTF8**, **UTF32**, **ASCII**, **BigEndianUnicode**, **Default** і **OEM**. За замовчуванням у **PowerShell** використовується кодування **Unicode**. Для збереження тексту в Windows-кодуванні слід вибирати значення **Default** (кодування поточної кодової сторінки ANSI), для збереження тексту в DOS-кодуванні – значення **OEM**
-Width	Вказує число знаків у кожному вихідному рядку
-Append	Записує вихідні дані в кінець існуючого файлу, а не заміщує його вміст
-NoClobber	Задає режим перезапису файлу. При вказівці цього параметра, якщо вихідний файл вже існує, він не буде перезаписуватись (за замовчуванням, якщо файл існує за вказаним шляхом, командлет ***Out-File*** перезаписує його без попередження). Якщо одночасно використовуються параметри ***-Append*** і ***-NoClobber***, вихідні дані записуються в кінець існуючого файлу

Наприклад, наступна команда збереже у файлі ***C:\help.txt*** детальний варіант вбудованої довідки за командлетом ***Get-Process*** (файл ***C:\help.txt*** буде створений у Windows-кодуванні):

PS F:\> Get-Help Get-Process -Detailed | Out-File -FilePath C:\help.txt -Encoding "Default"

3.4.4. Друк даних та придушення виведення

Дані можна вивести безпосередньо на принтер за допомогою командлета ***Out-Printer***. При цьому друк може здійснюватися як на принтері за замовчуванням (ніяких спеціальних параметрів для цього вказувати не треба), так і на довільному принтері (в цьому випадку коротке ім’я принтера має бути зазначено як значення параметра ***-Name***). Наприклад:

PS F: > Get-Process | Out-Printer -Name "Xerox Phaser 3500 PCL 6"

Командлет ***Out-Null*** служить для відкидання будь-яких своїх вхідних даних. Це може стати в нагоді для придушення виведення на екран непотрібних відомостей, отриманих як побічний ефект виконання будь-якої команди. Наприклад, під час створення каталогу командою ***mkdir*** на екран виводиться його вміст.

```
PS F:\> New-Item dir1 -Type "directory"


    Каталог: F:\


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----        05.05.2021     19:45                dir1
```

Якщо ви не бажаєте бачити цю інформацію, то результат виконання команди ***mkdir*** потрібно передати по конвеєру командлету ***Out-Null***:

```
PS F:\> New-Item dir2 -Type "directory" |  Out-Null
PS F:\>
```

У цьому випадку ніяких повідомлень на екран не виводиться.

3.5. Програмування сценаріїв командного рядка у Windows PowerShell

У багатьох наведених раніше прикладах ми вже використали різні числові і символьні літерали (константи), а також змінні **PowerShell**, зберігаючи в них результати виконання команд. Крім змінних у **PowerShell**, як у багатьох інших мовах програмування, підтримуються масиви, а також більш специфічні структури – асоціативні масиви (хеш-таблиці). Розглянемо ці елементи мови **PowerShell** більш докладно.

Практично в кожній мові програмування є можливість роботи з числами і символьними рядками, причому способи їх завдання можуть бути. **PowerShell** також підтримує цілі і дійсні числа, а також символьні рядки кількох видів.

Мова PowerShell підтримує всі основні числові типи платформи **.Net**: ***System.Int32***, ***System.Int64***, ***System.Double***. При цьому явно задавати тип чисел немає необхідності – система сама вибирає відповідний тип для зазначеного вами числа. Перевіримо тип декількох чисел, скориставшись для цього методом ***GetType***:

```
PS F:\> (10).GetType().FullName
System.Int32
PS F:\> (10.23).GetType().FullName
System.Double
PS F:\> (10+10.23).GetType().FullName
System.Double
```

У **PowerShell** передбачені спеціальні суфікси-множники для спрощення роботи з величинами, часто використовуваними системними адміністраторами: кілобайтами, мегабайтами і гігабайтами (табл. 3.5).

Табл. 3.5

Суфікси-множники в **PowerShell**

Суфікс-множник	Числовий множник	Приклад	Числове значення для прикладу
KB	1024	2 KB	2048
kb	1024	1 kb	1126.4
MB	1024*1024	3MB	3 145 728
mb	1024*1024	2.5mb	2 621 440
GB	1024*1024*1024	1GB	1 073 741 824
gb	1024*1024*1024	2.23gb	2 394 444 267.52

Наведемо приклади:

```
PS F:\> 1mb+10kb
1058816
PS F:\> 2GB+56MB
2206203904
```

У **PowerShell** можна оперувати числами в шістнадцятковому форматі, використовуючи для цього ті ж позначення, що і в С-подібних мовах програмування: перед числом вказується префікс ***0x***, а в записі числа можуть бути присутніми цифри і букви ***A***, ***B***, ***C***, ***D***, ***E*** і ***F*** (незалежно від регістру). Наприклад:

```
PS F:\> 0x10
16
PS F:\> 0xA
10
PS F:\> 0xcd
205
```

3.5.1. Символьні рядки

Усі символьні рядки в **PowerShell** є об'єктами типу ***System.String*** і являють собою послідовність 32-бітових символів у кодуванні **Unicode**. Довжина рядків не обмежена, вміст рядків не можна змінювати (можна тільки копіювати).

У **PowerShell** підтримується чотири види символьних рядків.

Рядки можуть задаватися послідовністю символів, укладених в одинарні або подвійні лапки:

```
PS F:\> 'Рядок в одинарних лапках'
Рядок в одинарних лапках
PS F:\> "Рядок в подвійних лапках"
Рядок в подвійних лапках
```

Рядки можуть містити будь-які символи (в тому числі символи розриву рядка і повернення каретки), крім відповідного одиночного закриваючого символу (одинарної або подвійної лапки). Рядок в одинарних лапках може містити подвійні лапки і навпаки:

```
PS F:\> 'Рядок в "одинарних" лапках'
Рядок в "одинарних" лапках
PS F:\> "Рядок в 'подвійних' лапках"
Рядок в 'подвійних' лапках
```

Якщо всередині рядка потрібно помістити символ, що обмежує цей рядок (тобто одинарні або подвійні лапки), то потрібно написати цей символ два рази поспіль:

```
PS F:\> 'Рядок в ''одинарних лапках'
Рядок в 'одинарних лапках
PS F:\> "Рядок в ""подвійних лапках"
Рядок в "подвійних лапках
```

Рядки в подвійних лапках є розширюваними. Це означає, що якщо всередині рядка в подвійних лапках трапляється ім'я змінної або інший вираз, який може бути обчислений, то в такий рядок підставляється значення цієї змінної або результат обчислення виразу. Наприклад:

```
PS F:\> $a = 123
PS F:\> "$a дорівнює $a"
123 дорівнює 123
```

Якщо ім'я змінної є всередині рядка в одинарних лапках, то ніякої підстановки значення змінної не відбувається:

```
PS F:\> '$a дорівнює $a'
$a дорівнює $a
```

За необхідності можна відключити розширення певної змінної всередині рядка в подвійних лапках. Для цього перед знаком ***$*** цій змінній потрібно вказати символ зворотного апострофа (`), наприклад:

```
PS F:\> "$a дорівнює $a"
123 дорівнює 123
```

Символи, що мають спеціальне значення, вставляються в рядки в подвійних лапках за допомогою escape-послідовностей, які в **Power-Shell** починаються з символу зворотного апострофа (`). В інших мовах програмування типу **C**, ***C#***, ***JScript*** або ***Perl*** для виділення спеціальних символів (escape-послідовностей) використовується зворотна коса риска (наприклад, ***\n*** або ***\t***). Розробники оболонки **PowerShell** прийняли рішення ввести інший символ для escape-послідовностей, щоб уникнути проблем під час використання символу \ як розділювача компонентів шляху у файловій системі **Windows** та інших просторах імен **PowerShell**.

Вставимо символ розриву рядка в рядок у подвійних лапках:

```
PS F:\> "Рядок в `nподвійних лапках"
Рядок в
подвійних лапках
```

Як бачите, на екран інформація виводиться у двох рядках. Якщо ж вставити escape-послідовність ***`n*** у рядок в одинарних лапках, то розриву рядка не відбудеться:

```
PS F:\> 'Рядок в `nодинарних лапках'
Рядок в `nодинарних лапках
```

Крім змінних, у розширюваних рядках можуть зазначатися так звані підвирази (subexpression) – обмежені символами ***$(...)*** фрагменти коду мовою **PowerShell**, які в рядках замінюються на результати обчислення цих фрагментів. Наприклад:

```
PS F:\> "3+2 дорівнює $(3+2)"
3+2 дорівнює 5
```

У **PowerShell** поряд зі звичайними рядками в одинарних і подвійних лапках підтримуються так звані автономні рядки (також відомі під ім’ям рядків типу «here-string»). Подібні рядки зазвичай використовуються для вставки в сценарій великих блоків тексту або під час генерації текстової інформації для інших програм, і мають такий вигляд:

@ <лапка> <розрив_рядка> блок тексту <розрив_рядка><лапка>@

Лапки можуть бути як одинарними, так і подвійними, при цьому сенс їх залишається тим же, що і для звичайних рядків: змінні і підвирази, які стоять всередині подвійних лапок, замінюються їх значеннями, а ті, які стоять всередині одинарних лапок, залишаються незмінними. Наприклад:

```
PS F:\> $a=@"
>> 1 Перший рядок
>> $(1+1) Другий рядок
>> "Третій рядок"
>> "@
PS F:\> $a
1 Перший рядок
2 Другий рядок
"Третій рядок"
```

```
PS F:\> $a=@'
>> 1 Перший рядок
>> $(1+1) Другий рядок
>> "Третій рядок"
>> '@
PS F:\> $a
1 Перший рядок
$(1+1) Другий рядок
"Третій рядок"
```

3.5.2. Змінні PowerShell

Як ми вже знаємо, імена змінних **PowerShell** завжди починаються зі знака долара (*$*). Змінні **PowerShell** не потрібно попередньо оголошувати або описувати, вони створюються при першому присвоєнні змінній значення. Якщо спробувати звернутися до неіснуючої змінної, то система поверне значення ***$Null***.

$Null, ***$True*** та ***$False***, є спеціальними змінними, визначеними в системі. Змінити значення цих змінних не можна.

Перевірити, чи визначена змінна, можна за допомогою командлета ***Test-Path***. Наприклад, наступна команда перевіряє, чи існує змінна ***MyVariable***:

```
PS F:\> Test-Path Variable:MyVariable
False
```

Список всіх змінних, визначених у поточному сеансі роботи, можна побачити, звернувшись до віртуального диску **PowerShell** ***Variable:*** за допомогою команди ***dir***:

```
PS F:\> dir Variable:

Name                           Value
----                           -----
$                              Variable:MyVariable
?                              True
^                              Test-Path
a                              1 Перший рядок...
args                           {}
ConfirmPreference              High
ConsoleFileName
DebugPreference                SilentlyContinue
```

Змінні оболонки – це набір змінних, які створюються, оголошуються оболонкою **PowerShell** і присутні за замовчуванням у кожному сеансі роботи. Змінні оболонки зберігаються протягом всього сеансу і доступні всім командам, сценаріями і додатків, які виконуються в цьому сеансі.

Підтримуються два види змінних оболонки.

Автоматичні змінні. У цих змінних зберігаються параметри стану оболонки **PowerShell**. Автоматичні змінні зберігаються і динамічно змінюються самою системою. Його користувачі не можуть (і не повинні) змінювати значення цих змінних. Наприклад, значенням змінної $ PID є ідентифікатор поточного процесу PowerShell.exe.

Змінні налаштувань. У цих змінних зберігаються налаштування активного користувача. Ці змінні створюються оболонкою **PowerShell** і заповнюються значеннями за замовчуванням. Користувачі можуть змінювати значення цих змінних. Наприклад, змінна ***$Maximum-HistoryCount*** визначає максимальне число записів у журналі сеансу.

У табл. 3.6 наведено короткий опис змінних оболонки.

Табл. 3.6

Змінні оболонки **PowerShell**

Змінна	Опис
$$	Містить останню лексему останнього отриманого оболонкою рядка
$?	Показує, чи успішно завершилася остання операція
$^	Містить першу лексему останнього отриманого оболонкою рядка
$_	У використанні в блоках сценаріїв, фільтрах і інструкції ***Where*** містить поточний об'єкт конвеєра
$Args	Містить масив параметрів, що передаються у функцію
$Error	Містить об'єкти, для яких виникла помилка під час обробки в командлеті
$ForEach	Звертається до ітератору в циклі ***ForEach***
$Home	Вказує домашній каталог користувача. Еквівалент конструкції ***%HomeDrive%%HomePath%*** в оболонці **cmd.exe**
$Input	Використовується в блоках сценаріїв, що знаходяться в конвеєрі
$PSHome	Показує ім'я каталогу, в якому встановлений **PowerShell**
$Host	Містить відомості про поточний вузол
$OFS	Використовується як розділювач під час перетворення масиву в рядок. За замовчуванням ця змінна має значення проміжку

Змінними оболонки можна користуватися так само, як і іншими видами змінних. Наприклад, наступна команда виведе на екран вміст домашнього каталогу **PowerShell**, шлях до якого зберігається в змінній оболонки ***$PSHome***:

```
PS F:\> dir $PSHome


    Каталог: C:\Windows\System32\WindowsPowerShell\v1.0


Mode                LastWriteTime         Length Name
----                -------------         ------ ----
d-----       17.07.2016      2:10                en-US
d-----       16.07.2016     14:47                Examples
d-----       24.01.2021     13:50                Modules
d-----       17.07.2016      2:10                ru
d-----       17.07.2016      2:10                ru-RU
d-----       16.07.2016     14:47                Schemas
d-----       16.07.2016     14:47                SessionConfig
-a----       16.07.2016     14:44          12825 Certificate.format.ps1xml
-a----       16.07.2016     14:44           5074 Diagnostics.Format.ps1xml
-a----       16.07.2016     14:44         138223 DotNetTypes.format.ps1xml
-a----       16.07.2016     14:44          10144 Event.Format.ps1xml
-a----       16.07.2016     14:44          25526 FileSystem.format.ps1xml
-a----       16.07.2016     14:44           9164 getevent.types.ps1xml
-a----       16.07.2016     14:44          91655 Help.format.ps1xml
-a----       16.07.2016     14:44         138625 HelpV3.format.ps1xml
-a----       15.09.2016     19:31         446976 powershell.exe
-a----       16.07.2016     14:44         206468 PowerShellCore.format.ps1xml
-a----       16.07.2016     14:44           4097 PowerShellTrace.format.ps1xml
-a----       29.10.2020      6:46         212992 powershell_ise.exe
-a----       16.07.2016     14:43            363 powershell_ise.exe.config
-a----       16.07.2016     14:44          55808 PSEvents.dll
-a----       16.07.2016     14:43         174592 pspluginwkr.dll
-a----       16.07.2016     14:44           2560 pwrshmsg.dll
-a----       16.07.2016     14:44          29184 pwrshsip.dll
-a----       16.07.2016     14:44           8458 Registry.format.ps1xml
-a----       16.07.2016     14:44         210376 types.ps1xml
-a----       16.07.2016     14:44          12282 typesv3.ps1xml
-a----       16.07.2016     14:44          16598 WSMan.Format.ps1xml
```

Призначена для користувача змінна створюється після першого присвоювання їй значення. Наприклад, створимо цілочисельну змінну ***$a***:

```
PS F:\> $a=1
PS F:\> $a
1
PS F:\> Test-Path Variable:a
True
PS F:\> dir Variable:a

Name                           Value
----                           -----
a                              1
```

Перевіримо, який тип має змінна ***$a***. Для цього можна скористатися командлетом ***Get-Member*** або методом ***GetType()***".

```
PS F:\> $a | Get-Member


   TypeName: System.Int32

Name        MemberType Definition
----        ---------- ----------
CompareTo   Method     int CompareTo(System.Object val
Equals      Method     bool Equals(System.Object obj),
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
```

```
PS F:\> $a.GetType().FullName
System.String
```

Отже, змінна ***$a*** зараз має тип ***System.Int32***. Дамо цій зміннійй інше значення (рядок) і знову перевіримо тип:

```
PS F:\> $a="aaa"
PS F:\> $a | Get-Member


   TypeName: System.String

Name             MemberType            Definition
----             ----------            ----------
Clone            Method                System.Object Clone(), Syst
CompareTo        Method                int CompareTo(System.Object
Contains         Method                bool Contains(string value)
CopyTo           Method                void CopyTo(int sourceIndex
EndsWith         Method                bool EndsWith(string value)
Equals           Method                bool Equals(System.Object o
GetEnumerator    Method                System.CharEnumerator GetEn
GetHashCode      Method                int GetHashCode()
GetType          Method                type GetType()
```

Можна також явно вказати тип змінної під час її визначення, вказавши у квадратних дужках відповідний атрибут типу. При цьому вираз, що стоїть у правій частині після знака рівності, буде перетворено (якщо це можливо) на такий тип. Наприклад, оголосимо цілочисельну змінну ***$a*** й дамо цій змінної символьне значення, яке можна перетворити до цілого типу:

```
PS F:\> [System.Int32]$a=10
PS F:\> $a="123"
PS F:\> $a
123
PS F:\> $a.GetType().FullName
System.Int32
```

Як бачите, рядок "*123"* був перетворений в ціле число ***123***. Якщо ж спробувати записати в змінну ***$a*** значення, яке не може бути перетворено в ціле число, то виникне помилка:

```
PS F:\> $a="aaa"
Cannot convert value "aaa" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:1
+ $a="aaa"
+ ~~~~~~~~
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformatio
   nMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

Замість явної вказівки .NET-типу змінної можна користуватися більш короткими псевдонімами типів відповідно до мови C#. Наприклад:

```
PS F:\> [int]$a=10
PS F:\> $a.GetType().FullName
System.Int32
```

3.5.3. Масиви у PowerShell

На відміну від багатьох мов програмування, в **PowerShell** не потрібно за допомогою будь-яких спеціальних символів вказувати початок масиву або його кінець, а також попередньо оголошувати масив.

Для створення і ініціалізації масиву можна просто привласнити значення його елементів. Значення, що додаються в масив, розділяються комою і відділяються від імені змінної (імені масиву) оператором присвоювання. Наприклад, вказана команда створить масив ***$a*** з трьох елементів:

```
PS F:\> $a=1,2,3
PS F:\> $a
1
2
3
```

Можна також створити й ініціалізувати масив, використовуючи оператор діапазону *(..)*. Наприклад, щоб створити й ініціалізувати масив ***$b***, що містить значення від ***10*** до ***14***, можна виконати таку команду:

```
PS F:\> $b=10..14
```

У результаті масив ***$b*** буде містити п’ять значень:

```
PS F:\> $b
10
11
12
13
14
```

Як ми вже бачили, для відображення всіх елементів масиву потрібно просто ввести його ім’я.

Довжина масиву (кількість елементів) зберігається у властивості ***Length***:

```
PS F:\> $a.Length
3
```

Для звернення до певного елементу масиву потрібно вказати його порядковий номер (індекс) у квадратних дужках після імені змінної. При цьому слід мати на увазі, що нумерація елементів у масиві завжди починається з нуля, тому для отримання значення першого елемента потрібно виконати таку команду:

```
PS F:\> $a[0]
1
```

Як індекс можна вказувати і негативні значення, при цьому відлік буде вестися з кінця масиву. Наприклад, індекс ***-1*** відповідатиме останньому елементу масиву:

```
PS F:\> $a[-1]
3
```

Щоб відобразити підмножину всіх значень у масиві, можна застосовувати оператор діапазону. Наприклад, щоб відобразити елементи з індексами від ***1*** до ***2***, потрібно ввести:

```
PS F:\> $a[1..2]
2
3
```

В операторі діапазону можна використовувати властивість ***Length***. Наприклад, для відображення елементів від індексу ***1*** до кінця масиву (останній елемент масиву має індекс ***Length-1***) можна виконати таку команду:

```
PS F:\> $a[1..($a.Length-1)]
2
3
```

Для зміни елемента масиву потрібно присвоїти нове значення елементу з відповідним індексом:

```
PS F:\> $a[0]=5
PS F:\> $a[1]=3.14
PS F:\> $a[2]="Привіт"
PS F:\> $a
5
3,14
Привіт
```

Останній приклад показує, що за замовчуванням масиви **PowerShell** можуть містити елементи різних типів, тобто є поліморфними. Визначимо тип нашого масиву ***$a***:

```
PS F:\> $a.GetType().FullName
System.Object[]
```

Отже, змінна ***$a*** має тип «масив елементів типу ***System.Object***». Можна створити масив з жорстко заданим типом, тобто такий масив, який містить елементи тільки одного типу. Для цього, як і у випадку зі звичайними скалярними змінними, необхідно вказати потрібний тип у квадратних дужках перед іменем змінної. Наприклад, ця команда створить масив 32-розрядних цілих чисел:

```
PS F:\> [int[]]$a=1,2,3,4
```

Якщо спробувати записати в цей масив значення, яке не можна перетворити до цілого типу, то виникне помилка:

```
PS F:\> $a[0]="aaa"
Cannot convert value "aaa" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:1
+ $a[0]="aaa"
+ ~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

У разі спроби звернутися до елемента, який виходить за межі масиву, виникне помилка. Наприклад:

```
PS F:\> $a.Length
4
PS F:\> $a[4]=5
Index was outside the bounds of the array.
At line:1 char:1
+ $a[4]=5
+ ~~~~~~~
    + CategoryInfo          : OperationStopped: (:) [], IndexOutOfRangeExc
   eption
    + FullyQualifiedErrorId : System.IndexOutOfRangeException
```

Подібні помилки пов’язані з тим, що масиви **PowerShell** базуються на .NET-масивах, що мають фіксовану довжину. Незважаючи на це, є спосіб збільшення довжини масиву. Для цього можна скористатися оператором конкатенації + або +=. Наприклад, вказана команда додасть до масиву ***$a*** два нові елементи зі значеннями ***5*** і ***6***:

```
PS F:\> [int[]]$a=1,2,3,4
PS F:\> $a
1
2
3
4
PS F:\> $a+=5,6
PS F:\> $a
1
2
3
4
5
6
```

Під час виконання оператора += відбувається таке:

1. **PowerShell** створює новий масив, розмір якого достатній для поміщення в нього всіх елементів.
2. Первісний вміст масиву копіюється в новий масив.
3. Нові елементи копіюються в кінець нового масиву.

Таким чином, ми насправді не додаємо новий елемент до масиву, а створюємо новий масив більшого розміру.

Видалити елемент з масиву не так просто, проте можна створити новий масив і скопіювати в нього всі елементи, крім непотрібного. Наприклад, наступна команда створить масив ***$b***, що містить всі елементи масиву, крім значення з індексом ***2***:

```
PS F:\> $b=$a[0,1+3..($a.Length-1)]
PS F:\> $b
1
2
4
5
6
```

Можна об'єднати два масиви в один за допомогою оператора конкатенації +. Наприклад:

```
PS F:\> $x=1,2
PS F:\> $y=3,4
PS F:\> $z=$x+$y
PS F:\> $z
1
2
3
4
```

Слід мати на увазі, що звичайний оператор присвоювання (=) діє на масиви за посиланням. Наприклад, створимо масив ***$a*** з двох елементів і присвоїмо цей масив змінній ***$b***:

```
PS F:\> $a=1,2
PS F:\> $b=$a
PS F:\> $b
1
2
```

Тепер змінимо значення першого елемента масиву ***$a*** й подивимося, що відбудеться з масивом ***$b***:

```
PS F:\> $a[0]="Нове значення"
PS F:\> $b
Нове значення
2
```

Як бачите, вміст масиву ***$b*** також змінився, оскільки змінна $b вказує на той же об'єкт, що і змінна ***$a***.

Для видалення масиву можна скористатися командлетом ***Remove-Item*** (псевдонім ***del***) і видалити змінну, яка містить потрібний масив, з віртуального диска ***variable:***. Наприклад:

```
PS F:\> $a
Нове значення
2
PS F:\> del variable:a
PS F:\> $a
PS F:\>
```

3.5.4. Хеш-таблиці (асоціативні масиви)

Крім звичайних масивів, у **PowerShell** підтримуються так звані **асоціативні масиви** (іноді їх також називають **словниками**) – структури для зберігання колекції ключів і їх значень, пов'язаних попарно.

Наприклад, можна використовувати прізвище людини як ключ, а його дату народження – як значення. Асоціативний масив забезпечує структуру для зберігання колекції імен і дат народження, де кожному імені зіставлена дата народження. Візуально масив асоційованих значень можна уявити як таблицю, що складається з двох стовпців, де перший стовпець є ключем, а другий – значенням.

Асоціативні масиви схожі на звичайні масиви в **PowerShell**, але замість звернення до вмісту масиву за індексом, до елементів даних асоціативного масиву можна звертатися за ключем. Використовуючи цей ключ, **PowerShell** повертає відповідне значення з асоціативного масиву.

Для зберігання вмісту асоціативного масиву в **PowerShell** використовується спеціальний тип даних – хеш-таблиця, оскільки така структура даних забезпечує швидкий механізм пошуку. Це дуже важливо, оскільки основним призначенням асоціативного масиву є забезпечення ефективного механізму пошуку.

На відміну від звичайного масиву, для оголошення і ініціалізації хеш-таблиць використовуються спеціальні літерали:

$ім'я_масиву = @{ключ! = елемент !; ключ2 = елемент2; ...}

Отже, кожному значенню хеш-таблиці потрібно прикріпити мітку (ключ), перед перерахуванням вмісту масиву потрібно поставити символи ***@{***, а завершити перерахування елементів символом ***}***. Ключі та значення поділяються знаком рівності (=). пари «ключ-значення» розділяються між собою крапкою з комою (***;***).

Створимо, наприклад, хеш-таблицю, в якій будуть зберігатися дані про одну людину (асоціативний масив з трьома елементами):

```
PS F:\> $user=@{Surname="Іванов";Name="Іван";Phone="11-11-11"}
PS F:\> $user

Name                           Value
----                           -----
Name                           Іван
Phone                          11-11-11
Surname                        Іванов
```

Тепер потрібно навчитися звертатись до її елементів. У **PowerShell** доступ до хеш-таблиць можливий двома способами: з використанням нотації властивостей або нотації масивів. Звернення з використанням нотації властивостей виглядає таким чином:

```
PS F:\> $user.Surname
Іванов
PS F:\> $user.Name
Іван
```

При цьому підході хеш-таблиця розглядається як об'єкт: ви вказуєте ім'я потрібної властивості і отримуєте відповідне значення. Звернення до асоціативного масиву з використанням нотації масивів відбувається так:

```
PS F:\> $user["Surname"]
Іванов
PS F:\> $user["Name"]
Іван
PS F:\> $user["Surname","Name"]
Іванов
Іван
```

Під час роботи з хеш-таблицею як з масивом можна отримувати значення відразу для декількох ключів.

Базовим типом для асоціативних масивів PowerShell є тип ***System.Collections.Hashtable***:

```
PS F:\> $user.GetType().FullName
System.Collections.Hashtable
```

У цьому типі визначені кілька властивостей і методів, які можна використовувати (нагадаємо, що повний список властивостей і методів можна отримати з допомогою командлета ***Get-Member***). Наприклад, у властивостях ***Keys*** і ***Values*** зберігаються всі ключі і все значення, відповідно:

```
PS F:\> $user.Keys
Name
Phone
Surname
PS F:\> $user.Values
Іван
11-11-11
Іванов
```

Давайте навчимося додавати елементи в хеш-таблицю, змінювати і видаляти їх. Додамо в хеш-таблицю ***$user*** дані про вік людини і про місто, де вона проживає:

```
PS F:\> $user.Age=30
PS F:\> $user

Name                           Value
----                           -----
Name                           Іван
Age                            30
Phone                          11-11-11
Surname                        Іванов


PS F:\> $user["City"]="Миколаїв"
PS F:\> $user

Name                           Value
----                           -----
City                           Миколаїв
Name                           Іван
Age                            30
Phone                          11-11-11
Surname                        Іванов
```

Таким чином, додаються елементи в асоціативний масив за допомогою простого оператора присвоювання з використанням нотації властивостей або масивів. Тепер змінимо значення вже наявного в масиві ключа. Робиться це також за допомогою оператора присвоювання:

```
PS F:\> $user["City"]="Херсон"
PS F:\> $user

Name                           Value
----                           -----
City                           Херсон
Surname                        Іванов
Name                           Іван
Age                            30
Phone                          11-11-11
```

Для видалення елемента з асоціативного масиву використовується метод ***Remove()***:

```
PS F:\> $user.Remove("Age")
PS F:\> $user

Name                           Value
----                           -----
City                           Херсон
Surname                        Іванов
Name                           Іван
Phone                          11-11-11
```

Можна створити порожню хеш-таблицю, не вказуючи жодної пари «ключ-значення». І потім заповнювати її послідовно по одному елементу:

```
PS F:\> $a=@{}
PS F:\> $a
PS F:\> $a.one=1
PS F:\> $a.two=2
PS F:\> $a

Name                           Value
----                           -----
one                            1
two                            2
```

Як і у випадку зі звичайними масивами, оператор присвоювання діє на хеш-таблиці за посиланням. Наприклад, після виконання зазначених команд змінні ***$a*** і ***$b*** вказуватимуть на один і той же об’єкт:

```
PS F:\> $b=$a
PS F:\> $b

Name                           Value
----                           -----
one                            1
two                            2
```

Змінивши значення одного з елементів в ***$a***, ми отримаємо ту ж зміну в ***$b***:

```
PS F:\> $a.one=3
PS F:\> $b

Name                           Value
----                           -----
one                            3
two                            2
```

3.6. Оператори у PowerShell

У мові **PowerShell** підтримується багато операторів, що дозволяють виконувати різні дії. Однією з особливостей операторів **PowerShell** є їх поліморфізм, тобто можливість застосовувати один і той же оператор до об'єктів різних типів. При цьому відмінність **PowerShell** від багатьох інших об'єктно-орієнтованих мов програмування полягає в тому, що поведінка операторів для основних типів даних (рядки, числа, масиви і хеш-таблиці) реалізується безпосередньо інтерпретатором, а не за допомогою того чи іншого методу об'єктів.

3.6.1. Арифметичні оператори

Основні арифметичні оператори, які підтримуються в **PowerShell**, наведені в табл. 3.7.

Табл. 3.7

Основні арифметичні оператори в **PowerShell**

Оператор	Опис	Приклад	Результат
+	Складає два значення	***2+4*** ***"aaa"+"bbb"*** ***1,2,3+4,5***	***6*** ***"aaabbb"*** ***1,2,3,4,5***
*	Перемножує два значення	***2*4*** ***"a"*3*** ***1,2,3*2***	***8*** ***"aaa"*** ***1,2,3,1,2,3***
-	Віднімає одне значення з іншого	***5-3***	***2***
/	Ділить одне значення на інше	***6/3*** ***7/4***	***2*** ***1.75***
%	Повертає залишок при цілочисельному діленні одного значення на інше	***7%4***	***3***

З точки зору поліморфної поведінки найбільш цікавими є оператори додавання і множення. Розглянемо їх більш детально.

Оператор додавання

Як уже згадувалося, поведінка операторів + і * для чисел, рядків, масивів і хеш-таблиць визначається самим інтерпретатором **Power Shell**. У результаті додавання або множення двох чисел виходить число. Результатом додавання (конкатенації) двох рядків є рядок. У результаті додавання двох масивів створюється новий масив, який є об'єднанням масивів.

А що станеться, якщо спробувати скласти об'єкти різних типів (наприклад, число з рядком)? У цьому випадку поведінка оператора буде визначатись типом операнда, що стоїть зліва. Слід запам'ятати так зване «правило лівої руки»: тип операнда, що стоїть зліва, задає тип результату дії оператора.

Якщо лівий операнд є числом, то **PowerShell** спробує перетворити правий операнд до числового типу. Наприклад:

```
PS F:\> 1+"12"
13
```

Як бачите, рядок ***"12"*** був перетворений до числа ***12***. Результат дії оператора додавання – число ***13***. Тепер зворотний приклад, коли лівий операнд є рядком:

```
PS F:\> "1"+12
112
```

Тут число ***12*** перетворюється до рядка ***"12"*** і в результаті конкатенації повертається рядок ***"112"***.

Якщо правий операнд можна перетворити до типу лівого операнда, то виникне помилка:

```
PS F:\> 1+"a"
Cannot convert value "a" to type "System.Int32". Error: "Input string was
not in a correct format."
At line:1 char:1
+ 1+"a"
+ ~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

Якщо операнд, що стоїть зліва від оператора складання, є масивом, то операнд, що стоїть праворуч, додається до цього масиву. При цьому створюється новий масив типу ***[object []]***, в який копіюється вміст операндів (це пов'язано з тим, що розмірність .NET-масивів фіксована). У процесі створення нового масиву всі обмеження на типи масивів, що додаються, будуть втрачені. Розглянемо приклад. Створимо спочатку масив цілих чисел:

```
PS F:\> $a=[int[]](1,2,3,4)
PS F:\> $a.GetType().FullName
System.Int32[]
```

Якщо спробувати змінити значення елемента цього масиву на який-небудь рядок, то виникне помилка, оскільки елементами масиву ***$a*** можуть бути тільки цілі числа:

```
PS F:\> $a[0]="aaa"
Cannot convert value "aaa" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:1
+ $a[0]="aaa"
+ ~~~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastFromStringToInteger
```

Додамо тепер до масиву ***$a*** ще один символьний елемент за допомогою оператора додавання:

```
PS F:\> $a=$a+"abc"
```

Знову спробуємо змінити значення першого елемента масиву ***$a***, записавши в нього символьний рядок:

```
PS F:\> $a=$a+"abc"
PS F:\> $a
1
2
3
4
abc
```

Як бачите, тепер помилка не виникає. Це пов'язано зі зміною типу масиву ***$a***:

```
PS F:\> $a.GetType().FullName
System.Object[]
```

Після збільшення масив ***$a*** отримує тип ***[object[]]***, і його елементами можуть бути об'єкти будь-якого типу.

Перейдемо тепер до складання хеш-таблиць. Як і у випадку зі звичайними масивами, під час додавання хеш-таблиць створюється новий асоціативний масив, в який копіюються елементи з масивів, що складаються. При цьому обидва операнди повинні бути хеш-таблицями (ніяке перетворення типів тут не підтримується). У новий масив спочатку копіюються елементи хеш-таблиці, що стоять зліва від оператора складання, а потім елементи хеш-таблиці, що стоять праворуч. Якщо ключове значення з правого операнда вже траплялося в лівому, то під час додавання виникне помилка.

Розглянемо приклад:

```
PS F:\> $left=@{a=1;b=2;c=3}
PS F:\> $right=@{d=4;e=5}
PS F:\> $sum=$left+$right
PS F:\> $sum

Name                           Value
----                           -----
c                              3
e                              5
a                              1
d                              4
b                              2
```

Нова результуюча хеш-таблиця як і раніше має тип ***System.Collections.Hashtable***:

```
PS F:\> $sum.GetType().FullName
System.Collections.Hashtable
```

Оператор множення

Оператор множення може діяти на числа, рядки і звичайні масиви (операція множення хеш-таблиць не визначена). При цьому праворуч від оператора множення обов'язково повинно знаходитися число (в іншому випадку виникне помилка).

Якщо зліва від оператора множення розташоване число, то операція множення виконується звичайним чином:

```
PS F:\> 6*5
30
```

Якщо лівим операндом є рядок, то він повторюється вказану кількість разів:

```
PS F:\> "abc"*3
abcabcabc
```

Аналогічним чином оператор множення діє на масиви:

```
PS F:\> $a=1,2,3
PS F:\> $a=$a*2
PS F:\> $a
1
2
3
1
2
3
```

Як і в випадку оператора складання, у результаті множення масиву на число створюється новий масив типу ***[object[]]*** потрібного розміру, і в нього копіюються елементи початкового масиву.

Оператори віднімання, ділення і залишку від ділення

Оператори віднімання (**-**), ділення (*/*) і залишку відділення (*%*) в **PowerShell** визначені тільки для чисел. Спеціального оператора цілочисельного ділення не передбачено: якщо діляться два цілих числа, то результатом буде дійсне число (тип ***System.Double***). Наприклад:

```
PS F:\> 123/4
30,75
```

Якщо потрібно округлити результат до найближчого цілого числа, то слід просто перетворити його до типу ***[int]***. Наприклад:

```
PS F:\> [int](123/4)
31
```

При цьому слід мати на увазі, що **PowerShell** у процесі перетворення дійсного числа в ціле використовує так зване «округлення Банкера»: якщо число є напівцілим (***0.5***, ***1.5***, ***2.5*** та ін.). то воно округлюється до найближчого парного числа. Таким чином, числа ***1.5*** і ***2.5*** округлюються до ***2***. а ***3.5*** і ***4.5*** округлюються до ***4***.

Якщо один з операндів є числом, а другий ні, то **PowerShell** намагається виконати перетворення до числового типу. Наприклад:

```
PS F:\> "123"/4
30,75
PS F:\> 123/"4"
30,75
```

Оператор присвоювання

Поряд з простим оператором присвоювання (=) в PowerShell підтримуються С-подібні складені оператори присвоювання: *+=*, ***-=***, ****=***, */=*, ***%=***.

3.6.2. Оператори перевірки на відповідність шаблону

Поряд з основними операторами порівняння в **PowerShell** є оператори перевірки символьних рядків на відповідність певним шаблоном. При цьому підтримуються два види шаблонів: **вирази з груповими символами** і **регулярні вирази**.

Шаблони з груповими символами

Раніше ми вже використали «підстановочні» (шаблонні) символи (. і ***?***) під час використання, скажімо, командлета ***dir***. Наприклад, команда ***dir *.doc*** виводить всі файли в поточному каталозі, що мають розширення ***doc***, – і будь-яке ім’я (шаблонний символ ***** замінює будь-яку кількість будь-яких символів).

У **PowerShell** підтримуються чотири види групових символів (табл. 3.8) і кілька операторів, які перевіряють рядки на відповідність шаблонами з підстановочними символами (були розглянуті вище).

Табл. 3.8

Групові символи в **PowerShell**

Символ	Опис	Приклад	Відповідає	Не відповідає
*	Будь-яка кількість довільних символів	a*	a ab abc	bc babe
?	Один довільний символ	a?b	acb a1b	a ab
[<симв 1> -<симв2>]	Діапазон символів від <симв1> до <симв2>	a[b-d]c	abc acc	aac ac
[<симв1> <симв2>...]	Будь-який символ зі вказаного набору	a[bc]c	abc ace	a adc

Шаблони з регулярними виразами

Регулярні вирази узагальнюють і розширюють концепцію шаблонів з груповими символами. Для завдання зразка використовуються літерали і метасимволи. Кожен символ, який не має спеціального значення в регулярних виразах, розглядається як буквальний і повинен точно збігатися під час пошуку. Наприклад, букви і цифри є літеральнимі символами. Метасимволи – це символи зі спеціальним значенням у регулярних виразах (табл. 3.9).

Табл. 3.9

Деякі метасимволи для регулярних виразів в **PowerShell**

Символ	Опис	Приклад	Відповідає	Не відповідає
.	Символ підстановки: відповідає будь-якому символу	a..	abc a34	a ab
*	Повторювач: означає нуль або більше попередніх символів, або класів символів	a.*b	ab abc awerbc	a bbc
?	Повторювач: означає нуль або один попередній символ, або клас символів	a?b	b ab cb	ac da
^	Положення в рядку: позначає початок рядка	^ab	abc abcd	sab acb
$	Положення в рядку: позначає кінець рядка	ab$	cdab ab	abc abb

Закінчення табл.

[<симв 1> <симв2> …]	Клас символів: позначає будь-який символ з вказаної множини	***a[bc]c***	***abc acc***	***a adc***
[^<симв 1> <симв2>]	Інвертований клас: позначає будь-який символ, який не входить у вказану множину	***a[^bc]c***	***adc a1c***	***abc acc***
[<симв 1>- <симв2>]	Діапазон: позначає будь-який символ з вказаного проміжку	***a[b-d]c***	***abc acc***	***aac ac***
<метаси мвол>	Виняток: визначає використання метасимволу як літералу	\\^***ab***	***c^ab ^ab***	***ab ^ac***

У **PowerShell** з регулярними виразами працюють оператори ***-match***, ***-notmatch*** і ***-replace***, а також їх варіанти.

Табл. 3.10

Оператори, що працюють з регулярними виразами

Оператор	Опис	Приклад	Результат
-match	Порівняння на збіг з урахуванням регулярний виразів у правому операнді	***"книга" -match "ни" "ніс" -match "[к-н]іс"***	***$True $True***
-notmatch	Порівняння на розбіжність з урахуванням регулярних виразів у правому операнді	***"книга" -notmatch "лкн"***	***$False***
-replace	Заміна або видалення символів у рядку – лівому операнді (ці оператори повертають змінений рядок). Якщо як правий операнд вказуються через кому два підрядки, то перший з них відповідає фрагменту, який потрібно змінити, а другий – рядку, який буде вставлений в результаті заміни. Якщо як правий операнд вказаний один підрядок, то він відповідає фрагменту рядка, який буде видалений	***"рід" -replace "д", "к" "рід" –replace "рі"***	***"рік" "д"***

3.6.3. Логічні оператори

Іноді всередині однієї інструкції необхідно перевірити відразу кілька умов. Оператори порівняння можна з'єднувати один з одним за

допомогою логічних операторів, наведених в табл. 3.11. Під час використання логічного оператора **PowerShell** перевіряє кожну умову окремо, а потім обчислює значення інструкцій цілком, пов'язуючи умови за допомогою логічних операторів.

Табл. 3.11

Логічні оператори в **PowerShell**

Оператор	Значення	Приклад (повертається значення $True)
-and	логічне І	***(10 -eq 10) -and (1 -eq 1)***
-or	логічне АБО	***(9 -ne 10) -or (3 -eq 4)***
-not	логічне НЕ	***-not (3 -gt 4)***
!	логічне НЕ	***!(3 -gt 4)***

3.7. Керуючі інструкції мови PowerShell

У мові **PowerShell**, як і в будь-якій іншій алгоритмічній мові, є елементи, що дозволяють виконати логічне порівняння і зробити різні дії залежно від його результату або дають можливість повторювати одну або кілька команд знову і знову.

3.7.1. Інструкція If ... ElseIf ... Else

Логічні порівняння лежать в основі практично всіх алгоритмічних мов програмування. У **PowerShell** за допомогою інструкції ***If*** можна виконувати певні блоки коду тільки в тому випадку, коли задана умова має значення ***$True*** (істина). Також можна задати одне або декілька додаткових умовних блоків. Відповідні їм умови будуть перевірятися, якщо всі попередні умови мали значення ***$False***. Нарешті, можна задати додатковий блок коду, який буде виконуватися в тому випадку, якщо жодна з умов не має значення ***$True***.

Синтаксис інструкції ***if*** у загальному випадку має такий вигляд:

If (умова1)
{блок_коду1}
[ElseIf (умова2)
{блок_коду2}]
[Else
{ блок_коду3}]

Під час виконання інструкції ***if*** перевіряється істинність умовного виразу ***умова1***. Якщо ***умова1*** має значення ***$True***, то виконується ***блок_коду1***, після чого **PowerShell** завершує виконання інструкції ***if***. Якщо ***умова1*** має значення ***$False***, то **PowerShell** перевіряє істинність умовного виразу ***умова2***. Якщо ***умова2*** має значення ***$True***, то виконується ***блок_коду2***, після чого **PowerShell** завершує

виконання інструкції ***if***. Якщо і ***умова1***, і ***умова2*** мають значення ***$False***, то виконується ***блок_коду3***, і виконання інструкції ***if*** завершується.

Наведемо приклад використання інструкції ***if***. Запишемо спочатку в змінну ***$a*** число ***10***:

```
PS F:\> $a=10
PS F:\> if ($a -eq 15) {
>> 'Значення $a дорівнює 15'
>> } else
>> {'Значення $a не дорівнює 15'}
Значення $a не дорівнює 15
```

З цього прикладу також видно, що в оболонці **PowerShell** в інтерактивному режимі можна виконувати інструкції, що складаються з декількох рядків (це може виявитися дуже корисним під час налагодження сценаріїв).

3.7.2. Цикл While

У **PowerShell** підтримуються кілька видів циклів. Найпростіший з них – цикл ***while***, в якому команди виконуються до тих пір, поки умова, що перевіряється, має значення ***$True***.

Інструкція ***while*** має такий синтаксис:

While (умова)
{блок_команд}

У процесі виконання інструкції ***while*** оболонка **PowerShell** обчислює розділ ***умова*** інструкції, перш ніж перейти до розділу ***блок_команд***. Умова в інструкції приймає значення ***$True*** або ***$False***. До тих пір, поки умова має значення ***$True***, **PowerShell** повторює виконання розділу ***блок_команд***.

Розділ ***блок_команд*** інструкції ***While*** містить одну або кілька команд, які виконуються кожен раз під час входу в цикл і його повторення.

Наприклад, зазначена інструкція ***while*** відображає числа від ***1*** до ***3***. Якщо змінна ***$val*** не була створена або була створена й ініціалізована значенням ***0***:

```
PS F:\> while ($val -ne 3)
>> {
>> $val++
>> $val
>> }
1
2
3
```

У цьому прикладі ***умова*** (значення змінної ***$val*** не дорівнює ***3***) має значення ***$True***, поки ***$val*** дорівнює ***0***, ***1*** або ***2***. З кожним повторенням циклу значення ***$val*** збільшується на ***1*** з використанням унарного оператора збільшення значення ++ (***$val***++). В останньому виконанні циклу значення ***$val*** стає рівним ***3***. При цьому умова, що перевіряється, приймає значення ***$False***, і цикл завершується.

3.7.3. Цикл Do ... While

Цикл ***Do ... While*** схожий на цикл ***While***, однак умова в ньому перевіряється не до блоку команд, а після:

Do {блок_команд} While (умова)

Наприклад:

```
PS F:\> $val=0
PS F:\> do {$val++; $val} while ($val -ne 3)
1
2
3
```

Цикл For

Інструкція ***For*** в **PowerShell** реалізує ще один тип циклів – цикл з лічильником. Зазвичай цикл ***For*** застосовується для проходження по масиву і виконання певних дій з кожним із його елементів. У **PowerShell** інструкція ***For*** використовується не так часто, як в інших мовах програмування, оскільки колекції об'єктів зазвичай зручніше обробляти за допомогою інструкції ***ForEach***. Однак якщо необхідно знати, з яким саме елементом колекції або масиву ми працюємо на цій ітерації, то цикл ***For*** може допомогти.

Синтаксис інструкції ***For***:

For (ініціалізація; умова; повторення) {блок_команд}

Класичний приклад:

```
PS F:\> for ($i=0; $i -lt 5; $i++) { $i }
0
1
2
3
4
```

3.7.4. Цикл ForEach

Інструкція ***ForEach*** дозволяє послідовно перебирати елементи колекцій. Найпростішим і найбільш часто використовуваним типом колекції, за якою проводиться переміщення, є масив. Зазвичай у циклі ***ForEach*** одна або кілька команд виконуються на кожному елементі масиву.

Особливістю циклу ***ForEach*** є те, що його синтаксис і робота залежать від того, де розташована інструкція ***ForEach***: поза конвеєром команд або всередині конвеєра.

Інструкція ForEach поза конвеєром команд

У цьому випадку синтаксис циклу ***ForEach*** має такий вигляд:

ForEach ($елемент in $колекція) {блок_команд}

У круглих дужках вказується колекція, за якою проводиться ітерація. Під час виконання циклу ***ForEach*** система автоматично створює змінну ***$елемент***. Перед кожною ітерацією в циклі цієї змінної присвоюється значення чергового елемента в колекції. У розділі ***блок_команд*** містяться команди, які виконуються на кожному елементі колекції.

Наприклад, цикл ***ForEach*** у наступному прикладі відображає значення в масиві з ім'ям ***$letterArray***:

```
PS F:\> $letterArray="a","b","c","d"
PS F:\> ForEach ($letter in $letterArray) {$letter}
a
b
c
d
```

Інструкція ***ForEach*** може також використовуватися спільно командлетами, що повертають колекції елементів. Наприклад:

```
PS F:\> $l=0; ForEach ($f in dir *.txt) {$l+=$f.Length}
PS F:\> $l
18008
```

Інструкція ForEach всередині конвеєра команд

Якщо інструкція ***ForEach*** з'являється всередині конвеєра команд, то **PowerShell** використовує псевдонім ***ForEach***, відповідний командлету ***ForEach-Object***. Тобто в цьому випадку фактично виконується командлет ***ForEach-Object***, і вже не потрібно вказувати частину інструкції (***$елемент in $колекція***), оскільки елементи колекції блоку команд надає попередня команда в конвеєрі.

Синтаксис інструкції ***ForEach***, застосовуваної всередині конвеєра команд, в найпростішому випадку виглядає таким чином:

команда* | *ForEach {блок_команд}

Розглянутий вище приклад з підрахунком сумарного розміру текстових файлів з поточного каталогу для такого варіанту інструкції ***ForEach*** матиме такий вигляд:

```
PS F:\> $l = 0; dir *.txt | ForEach { $l+=$_.Length }
PS F:\> $l
18008
```

У загальному випадку у псевдонімі ***ForEach*** може вказуватися не один блок команд, а три: початковий блок команд, середній блок команд і кінцевий блок команд. Початковий і кінцевий блоки команд виконуються один раз. А середній блок команд виконується кожен раз під час чергової ітерації за колекцією або масивом.

Синтаксис псевдоніма ***ForEach***, використовуваного в конвеєрі команд з початковим, середнім і кінцевим блоками команд, виглядає таким чином:

команда | ForEach (початковий_блок_команд} {середній_блок_команд) {кінцевий блок_команд}

Для цього варіанта інструкції ***ForEach*** наш приклад можна записати в такий спосіб:

```
PS F:\> dir *.txt | ForEach {$l=0} {$l+=$_.Length} {$l}
18008
```

3.7.5. Мітки циклів, інструкції Break і Continue

Інструкція ***Break*** дозволяє вийти з циклу будь-якого типу, не чекаючи закінчення його ітерацій. Розглянемо простий приклад:

```
PS F:\> dir *.txt | ForEach {$l=0} {$l+=$_.Length} {$l}
18008
```

Умовою циклу ***While*** в цьому випадку є логічна константа ***$True***, тому цей цикл не завершився б ніколи. Інструкція ***Break***, яка спрацьовує під час досягнення змінної ***$i*** значення ***3,*** дозволяє вийти з такого циклу.

Інструкція ***Continue*** здійснює перехід до наступної ітерації циклу будь-якого типу. Наприклад:

```
PS F:\> For ($i=0; $i -le 6; $i++) {If ($i -eq 3) {Continue} $i}
0
1
2
4
5
6
```

У мові **PowerShell** підтримується можливість негайного виходу або переходу до наступної ітерації не тільки для одиночного циклу, але і для вкладених циклів. Для цього циклам присвоюються спеціальні мітки, які вказуються на початку рядка перед ключовим словом, що задає цикл того чи іншого типу. Такі мітки потім можна використовувати під вкладеними циклами спільно з інструкціями ***Break*** або ***Continue***, вказуючи, на який саме цикл повинні діяти ці інструкції. Розглянемо простий приклад:

```
PS F:\> :outer while ($True) {
>> while ($True) {
>>     Break outer
>>   }
>> }
```

Тут інструкція ***Break*** здійснює вихід із зовнішнього циклу з міткою ***outer***. Якби мітка не було вказано, зовнішній цикл не завершився б ніколи.

3.7.6. Інструкція Switch

Інструкція ***switch***, яка об'єднує кілька перевірок умов всередині однієї конструкції, є в багатьох мовах програмування. Однак у мові **PowerShell** ця інструкція має потужні додаткові можливості:

– вона може використовуватися як аналог циклу, перевіряючи значення не єдиного елемента, а цілого масиву;

– вона може перевіряти елементи на відповідність шаблону з груповими символами або регулярними виразами;

– вона може обробляти текстові файли, використовуючи як елементи рядки з файлів, що перевіряються.

Розглянемо спочатку найпростішу форму інструкції ***Switch***, коли один скалярний вираз по черзі зіставляється з декількома умовами. Наприклад:

```
PS F:\> $a=3
PS F:\> switch ($a) {
>> "1"  {"Один"}
>> "2"  {"Два"}
>> "3"  {"Три"}
>> "4"  {"Чотири"}
>> }
Три
```

У цьому прикладі значення змінної ***$a*** послідовно порівнюється з числами ***1***, ***2*** ,***3*** і ***4***. У разі збігу виконується відповідний блок коду, зазначений у фігурних дужках (у нашому випадку просто виводиться рядок).

Якщо для значення, що перевіряється, справедливі кілька умов зі списку, то будуть виконані всі дії, зіставлені з цими умовами. Наприклад:

```
PS F:\> switch ($a) {
>> 1 {"Один"}
>> 2 {"Два"}
>> 3 {"Три"}
>> 4 {"Чотири"}
>> 3 {"Ще раз три"}
>> }
Три
Ще раз три
```

Якщо потрібно обмежитися тільки першим збігом, то слід застосувати інструкцію ***Break***:

```
PS F:\> Switch ($a) {
>> 1 {"Один"}
>> 2 {"Два"}
>> 3 {"Три"; break}
>> 4 {"Чотири"}
>> 3 {"Ще раз три"}
>> }
Три
```

У такому випадку перевірка умов всередині ***Switch*** переривається після знаходження першої відповідності. Якщо ж жодну відповідність не знайдено, то інструкція ***Switch*** не виконує ніяких дій. За допомогою ключового слова ***Default*** можна задати дію за замовчуванням, яка буде виконуватися в тому випадку, коли не знайдено жодної відповідності. Наприклад:

```
PS F:\> Switch (3) {
>> 1 {"Один"}
>> 2 {"Два"}
>> Default {"Ні один, ні два"}
>> }
Ні один, ні два
```

За замовчуванням в інструкції ***switch*** проводиться пряме порівняння з об'єктами, зазначеними в умови. Порівняння рядків при цьому проводиться без урахування регістрів символів, наприклад:

```
PS F:\> Switch ('абв') {
>> 'абв' {"Перше співпадіння"}
>> 'АБВ' {"Друге співпадіння"}
>> }
Перше співпадіння
Друге співпадіння
```

Якщо під час порівняння слід врахувати регістр символів, то потрібно вказати параметр ***-CaseSensitive***:

```
PS F:\> Switch -CaseSensitive ('абв') {
>> 'абв' {"Перше співпадіння"}
>> 'АБВ' {"Друге співпадіння"}
>> }
Перше співпадіння
```

Крім звичайного порівняння можна перевіряти елементи на відповідність шаблону з підстановочними символами. Для цього використовується перемикач ***-Wildcard***, наприклад:

```
PS F:\> Switch -CaseSensitive ('абв') {
>> 'абв' {"Перше співпадіння"}
>> 'АБВ' {"Друге співпадіння"}
>> }
Перше співпадіння
PS F:\> Switch -Wildcard ('абв') {
>> 'а*' {"Починається з 'а'"}
>> '*в' {"Закінується на 'в'"}
>> }
Починається з 'а'
Закінується на 'в'
```

Елемент (об'єкт), що перевіряється, доступний всередині інструкції ***Switch*** через спеціальну змінну *$_* (нагадаємо, що в змінній з такою назвою зберігається і поточний елемент, який передається по конвеєру від одного командлета до іншого). Наприклад:

```
PS F:\> Switch -CaseSensitive ('абв') {
>> 'абв' {"Перше співпадіння"}
>> 'АБВ' {"Друге співпадіння"}
>> }
Перше співпадіння
PS F:\> Switch -Wildcard ('абв') {
>> 'а*' {"Починається з 'а'"}
>> '*в' {"Закінується на 'в'"}
>> }
Починається з 'а'
Закінується на 'в'
PS F:\> Switch -Wildcard ('абв') {
>> 'а*' {"$_ починається з 'а'"}
>> '*в' {"$_ закінується на 'в'"}
>> }
абв починається з 'а'
абв закінується на 'в'
```

Перемикач ***-Regex*** дозволяє перевіряти елементи на відповідність шаблону, що містить регулярні вирази. Попередній приклад можна записати і в такий спосіб:

```
Switch -Regex ('абв') {
'^а' {"$_ починається з 'а'"}
'в$' {"$_ закінчується на 'в'"}
}
>>
абв починається з 'а'
абв закінчується на 'в'
```

Крім перевірок на простий збіг чи відповідність шаблону, інструкція ***Switch*** дозволяє виробляти більш складні перевірки, вказуючи замість шаблонів блоки коду мовою **PowerShell**. Значення, що перевіряється при цьому знову доступне через змінну *$_*. Розглянемо приклад:

```
PS C:\> Switch (10) {
>>     {$_ -gt 5} {"$_ більше 5"}
>>     {$_ -lt 20} {"$_ менше 20"}
```

```
>>    10 {"$_ дорівнює 10"}
>> }
>>
10 більше 5
10 менше 20
10 дорівнює 10
```

До теперішнього моменту всі значення, які ми перевіряли в інструкції ***Switch***, були скалярними величинами. Мова **PowerShell** допускає використання як значення, що перевіряється, масиву елементів, причому масиви можуть задаватися явно або виходити в результаті виконання будь-якої команди, скажімо, шляхом зчитування рядків з текстового файлу. Розглянемо приклад:

```
PS C:\> $arr=1,2,3,4,5,6
PS C:\> Switch ($arr) {
>>    {$_%3} {"$_ не ділиться на 3"}
>>    Default {"$_ ділиться на 3"}
>> }
>>
1 не ділиться на 3
2 не ділиться на 3
3 ділиться на 3
4 не ділиться на 3
5 не ділиться на 3
6 ділиться на 3
```

У такому випадку масив цілих чисел задається явним перерахуванням своїх елементів. Всі елементи масиву по черзі перевіряються всередині інструкції ***Switch***: якщо залишок від ділення поточного елемента на ***3*** не дорівнює нулю, то виводиться повідомлення ***"$_ не ділиться на 3"***. Де замість ***$_*** підставляється, як зазвичай, значення елемента, що перевіряється. В іншому випадку видається повідомлення ***"$ _ ділиться на 3"***.

Наведемо ще один приклад, коли масив перевірки елементів є результатом виконання команди **PowerShell**. Нехай нам потрібно дізнатися кількість файлів з розширеннями ***txt*** і ***log***, що знаходяться в системному каталозі Windows. Спочатку обнуляємо змінні-лічильники:

```
PS C:\> $txt=$log=0
```

Виконаємо тепер відповідну інструкцію ***switch***. Колекцію файлів з системного каталогу **Windows** отримаємо за допомогою команди ***dir $env:SystemRoot*** (нагадаємо, що у змінній середовища ***SystemRoot*** зберігається шлях до потрібного каталогу). Файли з розширеннями ***txt*** і

log будемо відбирати, перевіряючи на відповідність шаблонів з підстановочними символами (****.txt*** і ****.log***), і у відповідних блоках коду будемо збільшувати значення змінних ***$txt*** і ***$log***:

```
PS C:\> Switch -Wildcard (dir $env:SystemRoot) {
>> *.txt {$txt++}
>> *.log {$log++}
>> }
>>
```

Виведемо тепер значення змінних ***$txt*** і ***$log***:

```
PS C:\> "txt-файли: $txt log-файли: $log"
txt-файли: 21 log-файли: 156
```

3.8. Функції у PowerShell

До теперішнього моменту ми працювали переважно зі стандартними скомпільованими командлетами, функціональність яких ми не можемо змінити, оскільки їх вихідний код в оболонці **PowerShell** недоступний. Функції та сценарії – це два інших типи команд **PowerShell**, які можна створювати і змінювати на свій розсуд, керуючись мовою **PowerShell**.

Слід відразу звернути увагу, що функції в **PowerShell** мають деякі особливості порівняно з функціями в традиційних мовах програмування, оскільки **PowerShell** – це, в першу чергу, оболонка. У традиційних мовах функція, як правило, є аналогом методу об'єкта, а в **PowerShell** функція – це команда. Звідси відмінність у способах виконання функцій і передачі їм аргументів.

Зазвичай функції в традиційних мовах повертають єдине значення того чи іншого типу. «Значенням» функції **PowerShell** може бути цілий масив різних об'єктів, оскільки кожний вираз, що обчислюється у функції, розміщує свій результат у вихідний потік.

Крім того, функції в **PowerShell** діляться на кілька типів відповідно до їх поведінки всередині конвеєра команд.

Функція в **PowerShell** – це блок коду, який має назву і знаходиться в пам'яті до завершення поточного сеансу командної оболонки. Якщо функція визначається без формальних параметрів, то для її завдання досить вказати ключове слово **Function**, потім ім'я функції і список виразів, що складають тіло функції (цей список повинен бути записаний у фігурні дужки). Наприклад, створимо функцію ***MyFunc***:

```
PS C:\> Function MyFunc {"Всім привіт!"}
```

Для виклику цієї функції достатньо просто ввести її ім'я:

```
PS C:\> MyFunc
```

Всім привіт!

Відзначимо, що в багатьох мовах програмування у процесі виконання функції після її імені потрібно вказувати круглі дужки. У **PowerShell** цього робити не можна:

PS C:\> MyFunc()
An expression was expected after '('.
At line:1 char:8

Функції можуть працювати з аргументами, які передаються їм при запуску, причому підтримуються два варіанти обробки таких аргументів: за допомогою змінної ***$Args*** і шляхом завдання формальних параметрів.

3.8.1. Обробка аргументів функцій за допомогою змінної $Args

Функція в **PowerShell** має доступ до аргументів, з якими вона була запущена, навіть якщо під час визначення цієї функції не були задані формальні параметри. Всі аргументи, з якими була запущена функція, автоматично зберігаються в змінній ***$Args***. Іншими словами, в змінної ***$Args*** міститься масив, елементами якого є параметри функції. зазначені у процесі її запуску. Для прикладу додамо змінну ***$Args*** у нашу функцію ***MyFunc***:

PS C:\> Function MyFunc {"Привіт, $Args!"}

Оскільки змінна ***$Args*** поміщена у рядок у подвійних лапках, то під час запуску функції значення цієї змінної буде обчислене (розширене), і результат буде вставлений у рядок. Викличемо функцію ***MyFunc*** з трьома параметрами:

PS C:\> MyFunc Андрій Сергій Іван
Привіт, Андрій Сергій Іван!

Як бачите, три зазначених нами параметри (три елементи масиву ***$Args***) поміщені у вихідний рядок і розділені між собою проміжками. Можна змінити символ, що розділяє елементи під час їх підстановки, перевизначивши значення спеціальної змінної ***$OFS***:

PS C:\> Function MyFunc {
>> $OFS=","
>> "Привіт, $Args!"}
>>
PS C:\> MyFunc Андрій Сергій Іван
Привіт, Андрій,Сергій,Іван!

Зверніть увагу, що на відміну від традиційних мов програмування, функції в **PowerShell** є командами (це не методи об'єктів!), тому їх

аргументи вказуються через проміжок без додаткових круглих дужок і виділення символьних рядків лапками. Щоб наочно переконатися в цьому, запустимо функцію ***MyFunc*** таким чином:

PS C:\> MyFunc "Андрій","Сергій","Іван"
Привіт, System.Object[]!

Нагадаємо, що масиви в **PowerShell** задаються перерахуванням своїх елементів, а круглі дужки, що оточують який-небудь вираз, означають, що цей вираз має бути обчислено. Тому помилки під час такого виконання функції не виникає, однак результат її виконання виявляється зовсім іншим, адже в цьому випадку у функцію передаються не три символьних аргументи, а один аргумент, який є масивом з трьох елементів!

Оскільки змінна ***$Args*** є масивом, то всередині функції можна звертатись до окремих аргументів за їх порядковим номером (нагадаємо, що нумерація елементів масивів починається з нуля), а за допомогою методу Count визначати загальну кількість аргументів, переданих функції. Для прикладу створимо функцію SumArgs, яка буде повідомляти про кількість своїх аргументів і обчислювати їх суму:

PS C:\> Function SumArgs {Кількість аргументів: $($Args. Count)"
>> $n=0
>> For($i=0; $i -lt $Args.Count; $i++) { $n+=$Args[$i] }
>>"Сума аргументів: $n"}
>>

Запустимо функцію ***SumArgs*** з трьома числовими аргументами:

PS C:\> SumArgs 1 2 3
Кількість аргументів: 3
Сума аргументів: 6

Крім використання масиву ***$Args*** в ***PowerShell*** підтримується альтернативний підхід до обробки аргументів функцій – за допомогою завдання формальних параметрів.

3.8.2. Формальні параметри функцій

У **PowerShell**, як і в більшості інших мов програмування, під час опису функції можна задати список формальних параметрів, значення яких під час виконання функції будуть замінені значеннями фактично переданих аргументів.

Список формальних параметрів вказується в круглих дужках після імені функції. Визначимо, наприклад, функцію ***Subtract*** для

знаходження різниці двох своїх аргументів (зменшуваному відповідає параметр ***$From***, від'ємнику – параметр ***$Count***):

```
PS F:\> Function Substract($From, $Count) {$From-$Count}
```

Під час виконання функції ***Subtract*** її формальні параметри будуть замінені фактичними аргументами, які визначаються або за позицією в командному рядку, або за іменем.

Наприклад:

```
PS F:\> Substract 10 2
8
```

У цьому випадку відповідність формальних параметрів фактично переданих аргументів визначається за позицією: замість першого параметра ***$From*** підставляється число ***10***, замість другого параметра ***$Count*** підставляється число ***2***.

Під час вказівки аргументів можна використовувати імена формальних параметрів (порядок вказівки аргументів при цьому стає несуттєвим), наприклад:

```
PS F:\> Substract -From 10 -Count 2
8
PS F:\> Substract -Count 3 -From 5
2
```

Якщо двічі вказати ім'я одного і того ж параметра, то виникне помилка:

```
PS F:\> Substract -From 10 -From 2
Substract : Cannot bind parameter because parameter 'From' is specified more than once. To provide multiple values to
parameters that can accept multiple values, use the array syntax. For example, "-parameter value1,value2,value3".
At line:1 char:20
+ Substract -From 10 -From 2
+                    ~~~~~
    + CategoryInfo          : InvalidArgument: (:) [Substract], ParameterBindingException
    + FullyQualifiedErrorId : ParameterAlreadyBound,Substract
```

У процесі виконання виконанні функції можливий і третій варіант завдання аргументів, коли для деяких задаються імена, а деякі визначаються за позицією в командному рядку. При цьому діє такий алгоритм:

– все іменовані аргументи зіставляються відповідним формальним параметрам і видаляються зі списку аргументів;

– решта параметрів (безіменні або ті, що мають ім'я, якому не відповідає жоден формальний параметр) зіставляються формальним параметрам за позицією.

Наприклад:

```
PS F:\> Substract -From 10 2
8
PS F:\> Substract -Count 2 10
8
```

За замовчуванням функції **PowerShell**, як і інші команди, поводяться поліморфним чином. Наприклад, визначимо функцію ***Add***, що складає два своїх аргументи:

```
PS F:\> Function Add ($x, $y) {$x+$y}
```

Виконаємо цю функцію з аргументами-числами і аргументами-рядками:

```
PS F:\> Add 2 3
5
PS F:\> Add "2" "3"
23
```

У першому випадку функція ***Add*** повертає число ***5***, а в другому – рядок ***"23"***.

За необхідності під час оголошення функції можна явно задати тип формальних параметрів. Наприклад, визначимо функцію ***IntAdd***, яка буде складати два свої цілочисельні аргументи:

```
PS F:\> Function IntAdd([int] $x, [int] $y) {$x+$y}
```

Як і в попередньому випадку, викличемо цю функцію з аргументами-числами і з аргументами-рядками:

```
PS F:\> IntAdd 2 3
5
PS F:\> IntAdd "2" "3"
5
```

Як бачите, результат виходить один і той же, оскільки символьні аргументи перетворюються на цілочисельноий тип. Якщо перетворення виконати не вдається, то виникне помилка.

Під час виконання функції може бути вказано більшу кількість фактичних аргументів, ніж було задано формальних параметрів. При цьому «зайві» аргументи будуть поміщені в масив ***$Args***. Для ілюстрації розглянемо функцію ***ShowArgs*** з двома формальними параметрами:

```
PS F:\> Function ShowArgs($x,$y) {
>> "Перший аргумент: $x"
>> "Другий аргумент: $y"
>> "Інші аргументи: $Args" }
```

Викличемо функцію з чотирма аргументами:

```
PS F:\> ShowArgs 1 2 3 4
Перший аргумент: 1
Другий аргумент: 2
Інші аргументи: 3 4
```

Додаткові аргументи дійсно зберігаються в масиві ***$Args***.

У процесі оголошення формальних параметрів можна вказати значення, які будуть приймати ці параметри за замовчуванням (якщо явно не вказано відповідний фактичний аргумент). Наприклад, перевизначити функцію ***Add*** як функцію з двома формальними параметрами ***$x*** і ***$y***, які за замовчанням ініціалізуються числами ***2*** і ***3***:

```
PS F:\> Function Add($x=2,$y=3){$x+$y}
```

Якщо почати його використання без аргументів, то обидва параметри ініціалізуються значеннями за замовчуванням і функція повертає число ***5***:

```
PS F:\> Add
5
```

Якщо запустити функцію ***Add*** з одним аргументом, то вказане значення буде присвоєне параметру ***$x***, а параметр ***$y*** знову ініціалізується значенням за замовчуванням (***$y=3***):

```
PS F:\> Add 5
8
```

У результаті вказівки двох параметрів функція ***Add*** поверне їх суму:

```
PS F:\> Add 6 6
12
```

У функціях також можна використовувати параметри-перемикачі, які повинні мати тип ***SwitchParameter***, що має псевдонім ***[Switch]***. Значеннями перемикача можуть бути тільки ***$True*** або ***$False***, тому форматувати подібний формальний параметр не потрібно.

Для прикладу визначимо функцію ***MyFunc*** з одним параметром-перемикачем, що визначає поведінку функції:

```
PS F:\> Function MyFunk([Switch] $Recurse) {
>> if ($Recurse) {
>> "Рекурсивний варіант функції"
>> }
>> else
>> {
>> "Звичайний варіант функції"
>> }
>> }
```

Запустивши функцію MyFunc без параметра, ми отримаємо повідомлення, що вона працює в звичайному режимі:

```
PS F:\> MyFunk
Звичайний варіант функції
```

Якщо вказано параметр ***-Recurse***, то функція ***MyFunc*** буде працювати в альтернативному режимі:

```
PS F:\> MyFunk -Recurse
Рекурсивний варіант функції
```

3.8.3. Значення, що повертаються

У традиційних мовах програмування функція зазвичай повертає єдине значення певного типу. В оболонці **PowerShell** справа йде інакше – тут результати всіх виразів або конвеєрів, що обчислюються всередині функції, направляються у вихідний потік. Розглянемо, наприклад, функцію ***MyFunc***, в якій обчислюються три числові вирази:

```
PS F:\> Function MyFunk{1+2;3*3;12/4}
```

Як бачите, в цій функції явно не повертається жодне значення. Запустимо функцію ***MyFunc***:

```
PS F:\> MyFunk
3
9
3
```

На екран виводяться три рядки з результатами обчислень. Тепер запустимо цю функцію і збережемо результат її роботи в змінній ***$Result***:

```
PS F:\> $Result=MyFunk
```

Перевіримо тип змінної ***$Result***:

```
PS F:\> $Result.GetType().FullName
System.Object[]
```

Для того, щоб певне значення не потрапляло у вихідний потік, слід скористатись командлетом ***Write-Host***, який не додає рядки у вихідний потік.

```
PS F:\> Function MyFunc {
>> $name=Read-Host "Введіть своє ім'я"
>> Write-Host "Доброго дня, $name!"
>> $name
>> }
PS F:\> MyFunc
Введіть своє ім'я: Гліб
Доброго дня, Гліб!
Гліб
PS F:\> $Result=MyFunc
Введіть своє ім'я: Гліб
Доброго дня, Гліб!
PS F:\> $Result
Гліб
```

Як бачите, командлет ***Write-Host*** виводить рядок безпосередньо на консоль, а не у вихідний потік, тому в цьому випадку функція ***MyFunc*** повертає єдиний рядок – значення змінної ***$name***.

У мові **PowerShell** є інструкція ***Return***, що виконує негайний вихід з функції.

3.8.4. Функції всередині конвеєра команд

Уся ідеологія PowerShell побудована на застосуванні конвеєрів команд, і функції тут не є винятком: їх теж можна використовувати всередині конвеєрів. При цьому для прийому всередині функції вхідного потоку об’єктів, переданих від іншої команди, служить змінна ***$Input***. У цієї змінної буде міститися колекція вхідних об’єктів. Розглянемо приклад:

```
PS F:\> Function Sum {
>> $n=0
>> ForEach ($i in $Input) {$n+=$i}
>> $n
>> }
```

```
PS F:\> 1..10 | Sum
55
```

```
PS F:\> Function Sum2 {
>> $n=0
>> While ($Input.MoveNext()) {
>> $n+=$Input.Current
>> }
>> $n
>> }
```

Запустимо функцію ***Sum2*** і переконаємося, що вона видає той же результат, що і функція ***Sum***:

```
PS F:\> 1..10 | Sum2
55
```

Таким чином, написати функцію для роботи всередині конвеєра неважко (достатьо знати про призначення змінної ***$Input***). Однак є один нюанс. У результаті отримання даних від попередньої команди функція призупиняє конвеєр і запускається тільки один раз, коли сформована вся колекція вхідних елементів. Іншими словами, звичайна функція не підтримує повною мірою механізм конвеєризації, коли елемент обробляється в конвеєрі, не чекаючи створення наступних елементів. Для більш ефективної роботи всередині конвеєра функцію слід оформляти у вигляді фільтра.

3.8.5. Фільтри в PowerShell

Фільтр – це функція особливого виду, яка, перебуваючи всередині конвеєра, запускається для кожного елемента, що входить (звичайні

функції запускаються один раз для всієї сукупності елементів, сформованої попередньою командою конвеєра).

Синтаксично фільтри відрізняються від функцій лише тим, що замість ключового слова ***Function*** вказується слово ***Filter***:

Filter ім'я_фільтра (параметри) {блок_кода}

Однак алгоритми роботи звичайної функції і фільтра розрізняються, якщо вони знаходяться всередині конвеєра. У звичайній функції доступ до вхідних елементів конвеєра здійснюється через колекцію ***$Input***. У фільтрі ж визначена змінна ***$_***, що відповідає поточному елементу конвеєра, що проходить через цей фільтр.

Розглянемо приклад. Напишемо фільтр, який буде подвоювати числа, що проходять через нього по конвеєру. Такий фільтр буде складатися всього з одного виразу:

```
PS F:\> Filter Double {$_*2}
```

Перевіримо роботу цього фільтра:

```
PS F:\> 1..4 | Double
2
4
6
8
```

```
PS F:\> 1..4 | Double | Double
4
8
12
16
```

Функціональність фільтрів дуже схожа на функціональність командлета ***ForEach-Object***, який можна вважати безіменним фільтром.

3.9. Сценарії PowerShell

Сценарії **PowerShell** є текстовими файлами з розширенням ***ps1***, в яких записаний код (команди, оператори та інші конструкції) мовою **PowerShell**. На відміну від сценаріїв **WSH** і командних файлів інтерпретатора **cmd.exe**, сценарії **PowerShell** можна писати поетапно, безпосередньо в самій оболонці, переносячи потім готовий код у зовнішній текстовий файл. Такий підхід значно спрощує вивчення мови і налагодження сценаріїв, дозволяючи відразу бачити результат виконання окремих частин сценарію.

Створити файл зі сценарієм **PowerShell** можна різними способами:

– скористатися зовнішнім текстовим редактором (наприклад, стандартним Блокнотом Windows), вручну ввести потрібні команди і зберегти файл з розширенням ***ps1***;

– виконати потрібні команди в оболонці **PowerShell**, скопіювати їх з консолі в буфер Windows і вставити в текстовий файл, відкритий у зовнішньому текстовому редакторі (прийоми роботи з оболонки з

буфером Windows обговорювалися в розділі 2), і потім зберегти отриманий у результаті файл з розширенням ***ps1***;

– працюючи в оболонці **PowerShell**, включити за допомогою командлета ***Start-Transcript*** режим протоколювання команд. У результаті буде створено зовнішній файл з усіма командами, що запускаються в сеансі роботи. З цього файлу можна скопіювати потрібні команди в інший текстовий файл і зберегти його з розширенням ***ps1***;

– перебуваючи в оболонці **PowerShell**, оформити команди **PowerShell** у вигляді рядків (звичайних або автономних) і перенаправити за допомогою символів > і >> ці рядки в зовнішній файл з розширенням ***ps1***.

Скористаємося останнім із запропонованих варіантів і створимо в каталозі ***C:\Script*** найпростіший сценарій ***test.ps1***. який буде складатися з єдиного рядка:

"Цей рядок друкується зі сценарію PowerShell"

Тепер запишемо потрібний рядок (включаючи подвійні лапки) у файл ***test.ps1***:

```
PS F:\> '"Цей рядок друкується зі сценарію PowerShell"' > test.ps1
```

Спробуємо тепер запустити наш сценарій. Нагадаємо, що сценарії виконуються системою тільки в тому випадку, коли це дозволено поточною політикою виконання. За замовчуванням діє політика **Restricted**, яка повністю забороняє виконання сценаріїв **PowerShell**. Це зроблено з міркувань безпеки, оскільки у сценаріях може міститися шкідливий код, який може пошкодити систему або несанкціоновано скористатися одними даними.

Перевіримо активну політику виконання за допомогою командлета ***Get-ExecutionPolicy***.

```
PS F:\> Get-ExecutionPolicy
Restricted
```

Якщо у системі діє більш сувора політика безпеки (***Restricted*** або ***AllSigned***), потрібно встановити політику ***RemoteSigned***, яка дозволяє виконувати непідписані локальні сценарії:

```
PS F:\> Set-ExecutionPolicy RemoteSigned
```

Цей командлет може не спрацювати, якщо запускати його не від імені адміністратора. Для того, щоб дозволити виконання сценаріїв для поточного користувача, необхідно додати параметр ***–Scope*** зі значенням ***CurrentUser***:

```
PS F:\> Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

Під час запуску сценаріїв **PowerShell** шлях до файлу з кодом потрібно завжди вказувати явно, навіть якщо сценарій знаходиться в поточному каталозі, оскільки це запобігає можливому несанкціонованому запуску іншої виконуваної програми з аналогічним ім’ям, що знаходиться, наприклад, у системному каталозі. Тому запустимо сценарій таким чином (крапка відповідає поточному каталогу):

```
PS F:\> .\test.ps1
Цей рядок друкується зі сценарію PowerShell
```

При цьому можна навіть не вказувати розширення ***.ps1***. А можна вказати і повний шлях до файлу.

Сценарії **PowerShell** можна запускати безпосередньо з командного рядка інтерпретатора **cmd.exe** або за допомогою пункту **Виконати** меню **Пуск**. Для цього потрібно вказати повний шлях до цього сценарію як параметра програми **powershell.exe** (повний шлях до **powershell.exe** можна не вказувати).

Нагадаємо, що командні файли інтерпретатора **cmd.exe**, так само як і сценарії **WSH** мовами **VBScript** або **JScript**, можна було запускати з Провідника Windows, просто клацаючи мишею на значках цих сценаріїв. Зі сценаріями **PowerShell** цей метод не працює – якщо двічі клацнути мишею на значку сценарію **PowerShell**, то він не запуститься, а відкриється для редагування в Блокноті (з точки зору безпеки це правильний підхід, що дозволяє запобігти випадковому запуску сценарію).

3.9.1. Передача аргументів у сценарії

Розбір і обробка аргументів, переданих у сценарії, проводиться практично так само, як і у функціях (взагалі, сценарій – це фактично функція, яка знаходиться не в оперативній пам’яті, а на диску).

Аргументи вказуються після імені сценарію і розділяються між собою проміжками. Змінна ***$Args*** всередині сценарію містить масив, елементами якого є аргументи функції, зазначені під час її запуску. Для прикладу напишемо сценарій ***SumArgs.ps1***, який буде повідомляти кількість параметрів, з якими він запущений, і їх суму.

Файл **SumArgs.ps1**:

```
"Кількість аргументів: $($Args.count)"
$n=0
for($i=0; $i -lt $Args.Count; $i++) { $n+=$Args[$i] }
"Сума аргументів: $n"
```

```
PS F:\> .\SumArgs.ps1 4 7 8
Кількість аргументів: 3
Сума аргументів: 19
```

Як бачите, масив ***$Args*** у сценаріях має таке ж значення і обробляється так само, як і у функціях.

У сценаріях можна визначати формальні параметри, замість яких під час виконання будуть підставлятися фактичні аргументи, передані в сценарій. У функціях формальні параметри перераховувалися в круглих дужках після імені, тобто поза тілом функції. У сценаріях так вчинити не можна, бо тут весь вміст файлу є тілом сценарію, тому формальні параметри задаються за допомогою спеціальної інструкції ***Param***. Ця інструкція повинна бути найпершою командою у файлі, передувати їй можуть тільки порожні рядки і коментарі. Для прикладу напишемо сценарій ***Add.ps1*** з двома формальними параметрами, який буде виводити суму своїх аргументів.

Файл **Add.ps1**:

```
Param($x=2,$y=3)
$x+$y
```

Запустимо отриманий сценарій з аргументами і без них:

```
PS F:\> .\Add 10 20
30
PS F:\> .\Add
5
```

Все працює, як і очікувалося: якщо не вказано жодних аргументів, то всередині сценарію використовуються значення за замовчуванням.

У звичайному режимі вихід зі сценарію, як і з функції, відбувається після виконання останньої інструкції в ньому. Нагадаємо, що у функції можна було скористатися інструкцією ***Return*** для примусового завершення роботи. Аналог цієї інструкції для сценаріїв – інструкція ***Exit***, що дозволяє завершити роботу сценарію і, за необхідності, повернути певний код повернення. Код повернення може аналізуватися в зовнішніх програмах (наприклад, командних файлах або сценаріях **WSH**), що запускають сценарій **PowerShell**.

У мові **PowerShell** коментарі починаються зі знака *#*. Всі символи, що стоять після цього знака, розглядаються як частина коментаря і не інтерпретуються системою. Наприклад:

```
PS F:\> # Весь рядок - коментар. Вираз 2+2 не обчислюється
PS F:\> 2+2 # Коментар в кінці рядка. Вираз перед ним обчислюється
4
```

3.10. Доступ з PowerShell до зовнішніх об’єктів

Однією з основних завдань, для вирішення яких створювалася оболонка **PowerShell**, було отримання з командного рядка доступу до різних об’єктних структур, підтримуваних операційною системою **Windows**. Раніше до подібних об’єктів можна було звертатися або з

повноцінних додатків за допомогою інтерфейсу прикладного програмування (API), або зі сценаріїв **WSH**. У будь-якому випадку для використання зовнішніх об'єктів доводилося писати програмний код і вивчати їх структуру, що значно ускладнювало роботу з ними і перешкоджало широкому поширенню технологій автоматизації серед системних адміністраторів і користувачів Windows.

Для вирішення цієї проблеми в **PowerShell** були розроблені спеціальні командлети, що дозволяють в інтерактивному режимі з оболонки звертатися до об'єктів WMI, COM і .NET.

У **PowerShell** є командлет ***New-Object***, що дозволяє, зокрема, створювати екземпляри зовнішніх COM-об'єктів, вказуючи відповідний об'єкт як значення параметра ***-ComObject***. Наприклад, екземпляр COM-об'єкта з програмним ідентифікатором ***WScript.Shell*** створюється таким чином:

```
PS F:\> $Shell = New-Object -ComObject WScript.Shell
```

Виконуючи командлет ***New-Object***, інтерпретатор **PowerShell** отримує з системного реєстру шлях до файлів потрібної бібліотеки типів. Потім за допомогою цієї бібліотеки в пам'ять завантажується екземпляр запитуваного об'єкта, і його інтерфейси стають доступними для використання в **PowerShell**. Посилання на створений об'єкт зберігається в змінній: надалі, використовуючи цю змінну, ми отримуємо доступ до властивостей і методів об'єкта, а також до його вкладених об'єктів (якщо вони є).

Подивимося, які властивості і методи є у COM-об'єкта ***WScript. Shell***. Для цього скористаємося, як зазвичай, командлетом ***Get-Member***, передавши йому по конвеєру змінну ***$shell***, в якій збережене посилання на цей COM-об'єкт:

```
PS F:\> $Shell | Get-Member

   TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd35090}

Name                     MemberType            Definition
----                     ----------            ----------
AppActivate              Method                bool AppActivate (Variant, Variant)
CreateShortcut           Method                IDispatch CreateShortcut (string)
Exec                     Method                IWshExec Exec (string)
ExpandEnvironmentStrings Method                string ExpandEnvironmentStrings (string)
LogEvent                 Method                bool LogEvent (Variant, string, string)
Popup                    Method                int Popup (string, Variant, Variant, Variant)
RegDelete                Method                void RegDelete (string)
RegRead                  Method                Variant RegRead (string)
RegWrite                 Method                void RegWrite (string, Variant, Variant)
Run                      Method                int Run (string, Variant, Variant)
SendKeys                 Method                void SendKeys (string, Variant)
Environment              ParameterizedProperty IWshEnvironment Environment (Variant) {get}
CurrentDirectory         Property              string CurrentDirectory () {get} {set}
SpecialFolders           Property              IWshCollection SpecialFolders () {get}
```

Спробуємо скористатися яким-небудь методом COM-об'єкта ***WScript.Shell***. Наприклад, метод ***CreateShortcut*** дозволяє швидко створювати ярлики для папок і файлів. Для прикладу ми створимо на

робочому столі активного користувача ярлик ***PSHome.lnk*** для папки, в якій встановлена оболонка **PowerShell**.

Шлях до робочого столу можна визначити різними способами, наприклад, за допомогою завжди визначеної в **PowerShell** змінної ***$Home***, в якій зберігається шлях до папки активного користувача:

```
PS F:\> $lnk = $Shell.CreateShortcut("$Home\Desktop\PSHome.lnk")
```

Використання об'єктів .NET

Платформа .NET побудована таким чином, що для звернення до тих чи інших об'єктів потрібно попередньо завантажити в операційну пам'ять відповідну збірку (assembly) – динамічну бібліотеку певного виду. Найбільш часто використовувані збірки завантажуються в **PowerShell** автоматично, їх список можна побачити за допомогою статичного методу ***getAssemblies()*** таким чином:

```
PS F:\> [AppDomain]::CurrentDomain.getAssemblies()

GAC    Version        Location
---    -------        --------
True   v4.0.30319     C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscorlib.dll
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.ConsoleHost\v4.0_3.0.0.0__31bf3856ad364e35\Micro
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System\v4.0_4.0.0.0__b77a5c561934e089\System.dll
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Core\v4.0_4.0.0.0__b77a5c561934e089\System.Core.dll
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\System.Ma
True   v4.0           C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Management.Automation.resources\v4.0_3.0.0.0_ru_31bf3856ad364e
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.Management.Infrastructure\v4.0_1.0.0.0__31bf3856ad364e35\Mi
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Xml\v4.0_4.0.0.0__b77a5c561934e089\System.Xml.dll
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Management\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Management.dl
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.DirectoryServices\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Direct
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Numerics\v4.0_4.0.0.0__b77a5c561934e089\System.Numerics.dll
True   v4.0           C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.ConsoleHost.resources\v4.0_3.0.0.0_ru_31bf3856ad
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\mscorlib.resources\v4.0_4.0.0.0_ru_b77a5c561934e089\mscorlib.resource
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_64\System.Data\v4.0_4.0.0.0__b77a5c561934e089\System.Data.dll
False  v4.0.30319
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Security\v4.0_3.0.0.0__31bf3856ad364e35\Microsof
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_64\System.Transactions\v4.0_4.0.0.0__b77a5c561934e089\System.Transactions.
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Configuration\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Configurat
False  v4.0.30319     C:\Program Files\WindowsPowerShell\Modules\PSReadline\1.2\Microsoft.PowerShell.PSReadLine.dll
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.CSharp\v4.0_4.0.0.0__b03f5f7f11d50a3a\Microsoft.CSharp.dll
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Commands.Utility\v4.0_3.0.0.0__31bf3856ad364e35\
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Configuration.Install\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Co
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Commands.Management\v4.0_3.0.0.0__31bf3856ad364e
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Windows.Forms\v4.0_4.0.0.0__b77a5c561934e089\System.Windows.Fo
True   v4.0.30319     C:\Windows\Microsoft.Net\assembly\GAC_MSIL\System.Drawing\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Drawing.dll
True   v4.0           C:\Windows\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.Security.resources\v4.0_3.0.0.0_ru_31bf3856ad364
False  v4.0.30319
False  v4.0.30319
```

Наприклад, для звернення до об'єктів ***WinForms*** потрібно за допомогою методу ***LoadWithPartialName*** завантажити збірку, що підтримують ці об'єкти (результат виконання методу приводиться до типу ***void*** для придушення виведення на екран непотрібної інформації):

```
PS F:\> [void][System.Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
```

Будь-який графічний додаток повинен мати головну форму – об'єкт типу ***Windows.Forms.Form***. Ми збережемо цю форму в змінній ***$form***:

```
PS F:\> $form = New-Object Windows.Forms.Form
```

У заголовок форми (властивість ***Text*** об'єкта ***$form***) запишемо рядок ***"Перша форма"***:

```
PS F:\> $form.Text = "Перша форма"
```

Тепер створимо кнопку (об'єкт типу ***Windows.Forms.Button***) з написом ***"Натисни!"***:

```
PS F:\> $button = New-Object Windows.Forms.Button
PS F:\> $button.Text = "Натисни!"
```

Визначимо тепер дію, яка виконуватиметься під час натискання кнопки. Для цього потрібно написати обробник події ***Click*** кнопки (тобто вказати, які команди повинні виконатися у результаті натискання на кнопку). Оброблювач подій – це спеціальний метод з назвою ***Add_подія***. Нехай у нашому випадку натискання кнопки буде призводити до закриття форми (метод ***Close*** об'єкта ***$form***):

```
PS F:\> $button.Add_Click({$form.Close()})
```

Тепер помістимо кнопку на форму за допомогою методу ***Add*** колекції ***Controls*** об'єкта ***$form***:

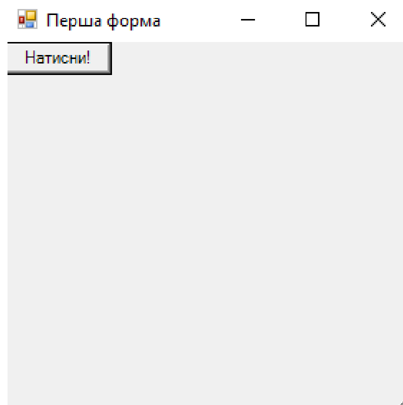
```
PS F:\> $form.Controls.Add($button)
```

Для того, щоб наша форма стала активною під час відображення, потрібно визначити обробник події ***Shown***, помістивши в цей обробник виклик методу ***Activate*** об'єкта ***$form***:

```
PS F:\> $form.Add_Shown({$form.Activate()})
```

Тепер залишилося викликати метод ***ShowDialog*** для виведення нашої форми на екран:

```
PS F:\> $form.ShowDialog()
```



Список використаних джерел

1. Книттель Б. Windows 7. Скрипты, автоматизация и командная строка. СПб. : Питер, 2012. 764 с.

2. У. Станек. Командная строка Microsoft Windows. Справочник администратора. СПб. : Русская редакция, 2009. 480 с.

3. Станек У. Windows PowerShell 2.0. Справочник администратора / Пер. с англ. – М. : Издательство «Русская редакция» ; СПб. : БХВ-Петербург ; 2010, – 416 с.

4. Попов А. В. Введение в Windows PowerShell. – СПб. : БХВ-Петербург, 2009. – 464 с.

5. Харалсон Д. Microsoft Windows Server 2003: администрирование из командной строки. М. : Кудиц-Образ, 2004. 576 с.

6. Попов А., Шикин Е. Администрирование Windows с помощью WMI и WMIC. СПб. : БХВ-Петербург, 2004. 752 с.

Для нотаток

Для нотаток

Навчальне видання

Горбань
Гліб Валентинович

Засоби автоматизації завдань в операційній системі Windows

Навчальний посібник

Редактор *А. Брмус*.
Технічний редактор *О. Петроченко*.
Комп'ютерна верстка, дизайн обкладинки *Д. Кардаш.*
Друк *С. Волинець*. Фальцювально-палітурні роботи *О. Мішалкіна.*

Підп. до друку 07.09.2021.
Формат 60х84$^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк ризограф.
Ум. друк. арк. 9,76. Обл.-вид. арк. 6,17.
Тираж 300 пр. Зам. № 6317.

Видавець і виготовлювач: ЧНУ ім. Петра Могили.
54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50–03–32, 8 (0512) 76–55–81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 6124 від 05.04.2018.