Pliki i katalogi

K. Barteczko 2009 -2017

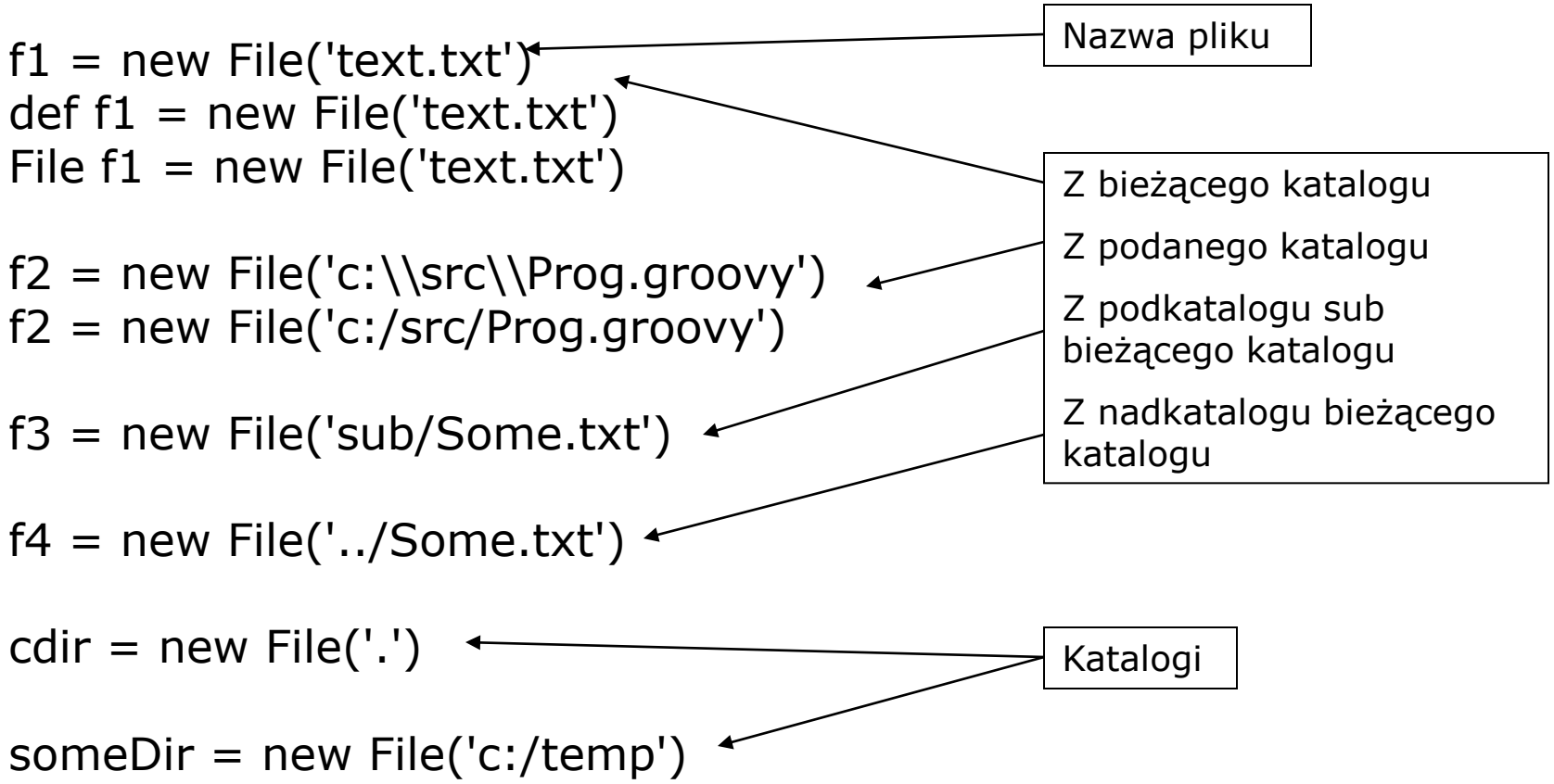
Pojęcie pliku

Plik - to ciąg bajtów zapisanych na dysku lub w innej fizycznie trwałej formie.

- po co są pliki
  - separacja danych od kodu,
  - utrwalanie i przenoszenie danych
- znaczenie bajtów
  - pliki binarne
  - pliki tekstowe
- strony kodowe
  - Unicode a inne systemy kodowania
  - domyślna strona kodowa
- kodowanie i dekodowanie

Obiekty typu File

Pliki i katalogi mogą być reprezentowane przez obiekty typu File.



Obiekty typu File mogą oznaczać istniejące lub nieistniejące pliki (wtedy plik jest tworzony dopiero w momencie zapisywania do niego) albo istniejące lub nieistniejące katalogi.

Path i Files w Javie a Groovy

W Javie 7 pojawiła się klasa java.nio.Files. To jest calkiem inna klasa niż klasa File, bo w przeciwieństwie do tej ostatniej zapewnia znacznie lepszą reprezentację współczesnych systemów i obiektów plikowych (m.in. większą liczbę atrybytów obiektów plikowych, obsługę tzw. symbolicznych linków). I - w przeciwieństwie do klasy File - dostarcza metod wejścia-wyjścia dla plików.
Większość metod klasy Files ma argumenty typu Path.
Path reprezentuje w sposób ogólny i uniwersalny (niezależny od konkretnego systemu plikowego) ścieżki obiektów plikowych (plików, katalogów itp.).

Groovy już od dawna zapewniał proste działania na plikach poprzez dodanie wielu metod do klasy File, ale mając na uwadze zgodność z Javą dodano te same metody od Path.

Tu będziemy mówić o przetwarzaniu plików głównie na przykładach użycia metod klasy File.

Czytanie plików tekstowych

```
def file = new File(...)

def cont = file.getText()

def cont = file.text  // skrót

def lineList = file.readLines()

file.eachLine { line ->
    // process line
}

file.splitEachLine(sep) { tokenListInLine ->
  // process token list
}
```

Wczytanie całego pliku naraz

Uzyskanie listy wierszy

Przetwarzanie wiersz po wierszu w domknięciu

Przetwarzanie z rozbiorem wierszy

Przykład 1

```
def file = new File('test1.txt')

def cont = file.text
if (cont.contains('23-10-15'))
  println 'Number found'

def lines = file.readLines()
println 'There are ' + lines.size() + ' lines.'
lines.eachWithIndex { line, i ->
  println "${i+1} $line"
}
println 'List of lines starting with "J"'
jlist = []
file.eachLine { if (it.startsWith('J')) jlist << it }
println jlist
```

```
Jan Kowalski      23-10-15
Joanna Malinowska          33-19-29
Stefan Nowak      17-16-89
```

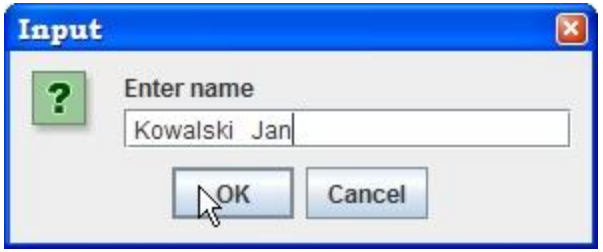
Result:

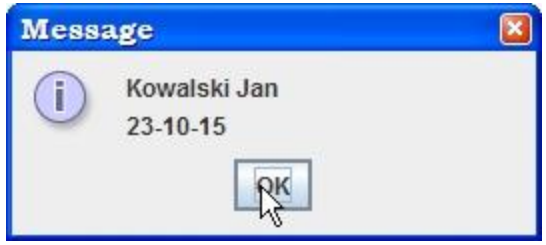
```
Number found
There are 3 lines.
1 Jan Kowalski    23-10-15
2 Joanna Malinowska        33-19-29
3 Stefan Nowak    17-16-89
List of lines starting with "J"
[Jan Kowalski     23-10-15, Joanna Malinowska        33-19-29]
```

Przykład 2

```
import static javax.swing.JOptionPane.*;

def file = new File('test1.txt')
def map = [:]
file.splitEachLine('\t') { tokens ->
   name = tokens[0].tokenize()
   map[name[1]+' '+name[0]] = tokens[1]
}
while ((inp = showInputDialog('Enter name')) != null) {
  name = inp.tokenize().join(' ')
  num = map[name] ?: 'Not found'
  showMessageDialog(null, "$name\n$num")
}
```





Przetwarzanie wierszy pliku

Oprócz metody readLines() są też inne sposoby uzyskania zestawu wierszy pliku. W kontekście file = new File(....)

arr = file as String[] // arr jest tablicą wierszy pliku

list = file.collect { it } // lista wierszy

Metoda collect jest odpowiednikiem "map" (w triadzie filter-map-reduce), więc możemy od razu uzyskiwane wiersze modyfikować, np. jeśli plik zawiera ścieżki z separatorem Windowsowym – już przy czytaniu zamienić separator na standardowy:

```
def f = new File('test2.txt')
lines = f.collect { it.replace('\\', '/') }
lines.each { println it }
```

```
D:\work\gallery-6.jpg
D:\Temp\logonbackup
```

```
D:/work/gallery-6.jpg
D:/Temp/logonbackup
```

Pliki tekstowe jako obiekty iterowalne

A elementem w każdej iteracji jest wiersz.

```
Zatem zamiast eachLine, mozna użyć each:

file.each { line -> .... }

I można też używac metod find i findAll:

file.find { it.startsWith('monday') } // wiersz zaczynający się od monday

file.findAll { it.contains('Poland')} // lista wierszy zawierających tekst
```

Należy przy tym jednak pamiętac, że pliki będą wczytywane w domyślnej stronie kodowej. Zob. dalej o problemie kodowania.

Zapisywanie plików tekstowych

```
def file = new File(...)
// Metody nadpisujące (lub tworzące nowy plik)
file.setText(txt)
file.text = txt  // skrót
file.write(txt)
// Metoda dopisujące
file.append(txt)
// Operator <<
file << txt
```

(tworzy nowy jeśli nie istnieje, dopisuje jeśli istnieje)

Przykład zapisywania plików

```
def file = new File('test2.txt')

def lines = ['aaa', 'bbb']
def cont = lines.join('\n')

file.text = cont  // creating and writing

showFile file

file.append('\nccc') // appending
file.append(111)     // can append any Object
showFile file

file << '\nxyz '  << [1, 7, 6]
showFile file

// overwriting
file.write('new content') // can write only String
showFile file

file.text = 'Again new content'
showFile file

def file2 = new File('out2.txt')
file2 << file.text // quick copy
showFile file2
```

Output:

```
Now test2.txt content is:
aaa
bbb
Now test2.txt content is:
aaa
bbb
ccc111
Now test2.txt content is:
aaa
bbb
ccc111
xyz [1, 7, 6]
Now test2.txt content is:
new content
Now test2.txt content is:
Again new content
Now out2.txt content is:
Again new content
```

```
def showFile(file) {
  println 'Now ' + file.name +
          ' content is:\n' + file.text
}
```

Problem końca wiersza

Normalnie LF (\n). Różne platformy różnie (LF, CRLF, CR).

Niezależnie od platformy:
1) używać System.getProperty('line.separator')
2) normalizować i denormalizować napisy wielowierszowe

```
def f1 = new File('normalized.txt')
def f2 = new File('platform.txt')
def txt = 'a\nb\nc'
f1.text = txt
f2.text = txt.denormalize()
println 'Normal separator is: ' + '\n'.bytes
println 'Platform line separator is ' +
         System.getProperty('line.separator').bytes
println 'f1 byte cont is: ' + f1.text.bytes
println 'f2 byte cont is: ' + f2.text.bytes
def norm = f2.text.normalize()
println 'Normalized f2 byte cont: ' + norm.bytes

Output:
Normal separator is: [10]
Platform line separator is [13, 10]
f1 byte cont is: [97, 10, 98, 10, 99]
f2 byte cont is: [97, 13, 10, 98, 13, 10, 99]
Normalized f2 byte cont: [97, 10, 98, 10, 99]
```

Pobór wartości bajtów.

Widać je tu - to są kody znaków (CR = 13, LF =10 (0D0A)

Bajty z pliku można też pobierać metodą eachByte.

Dekodowanie - kodowanie

Normalnie czytanie, pisanie plików używa domyślnej strony kodowej. Czasem to nie wystarcza

Kodowanie: Unicode -> wybrana strona kodowa

Dekodowanie: wybrana strona kodowa -> Unicode

```
def file = new File(...)
def cont = file.getText(charset) // dekodowanie
list = file.readLines(charset)   // dekodowanie
file.eachLine(charset) { .. }    // dekodowanie
file.write(txt, charset)         // kodowanie
file.append(txt, charset)        // kodowanie
```

Przykładowe charsety: 'ISO-8859-1', 'cp1250', 'UTF-8'

Zobaczyć wszystkie:

```
import java.nio.charset.*;
def chs = Charset.availableCharsets()
chs.each {
  println it.key + ' ' + it.value.aliases()
}
```

Przykład: użyteczne narzędzie

Uniwersalny kawałek kodu do zmiany kodowania plików:

```
def changeEncoding(inFile, cpin, outFile, cpout) {
    def cont = inFile.getText(cpin)
    outFile.setText(cont, cpout)
}
```

Przykładowe wywołanie:

```
changeEncoding(new File('windows-1250.html'), 'cp1250',
                    new File('iso-8859-2.html'), 'ISO-8859-2')
```

Pliki binarne

File f …

tablica_bajtow = f.bytes // (czytanie - skrót od f.getBytes())

f.bytes = tablica_bajtow // zapis – skrot od f.setBytes(…)

Szybkie kopiowanie:

out.bytes = input.bytes

Dodatkowo metody:

eachByte (Closure)

eachByte(int bufLen, Closure)

// tu closure ma dwa parametry – przeczytane bajty w tablicy (bufor o długości bufLen) oraz aktualnie przeczytaną liczbę bajtów.

append(byte[])

Pojęcie o strumieniach

Strumień - ogólniejsza abstrakcja odczytu-zapisu danych.

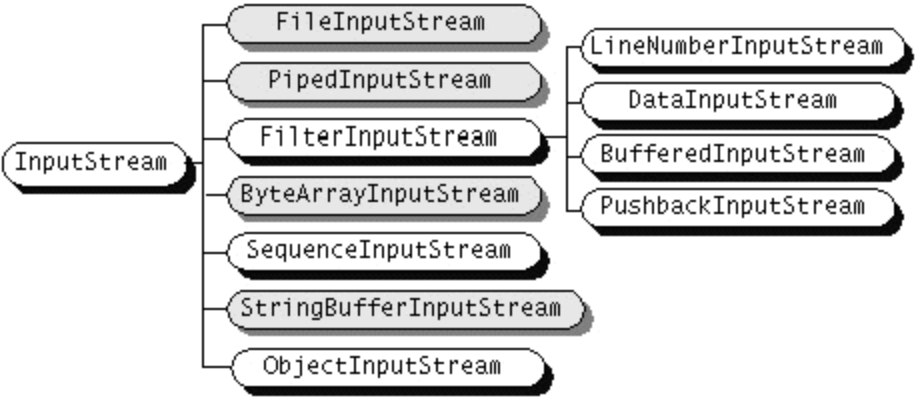
Źródła bądź odbiorniki danych mogą być różnorodne: plik, pamięć operacyjna, potoki, URL, gniazdo sieciowie.

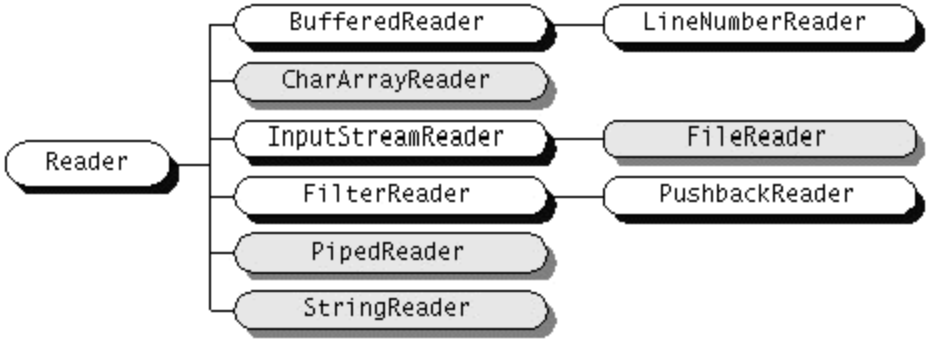
Przy odczycie - zapisie - można dokonywać transformacji danych.

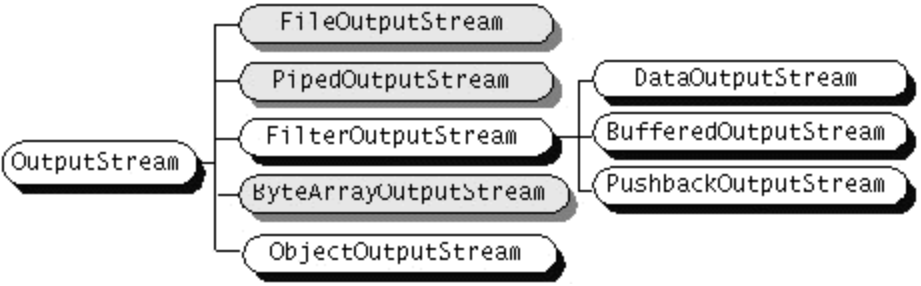
Strumienie są reprezentowane przez klasy strumieniowe, które tworzą hierarchie.
Początkowe klasy tych hierarchii:

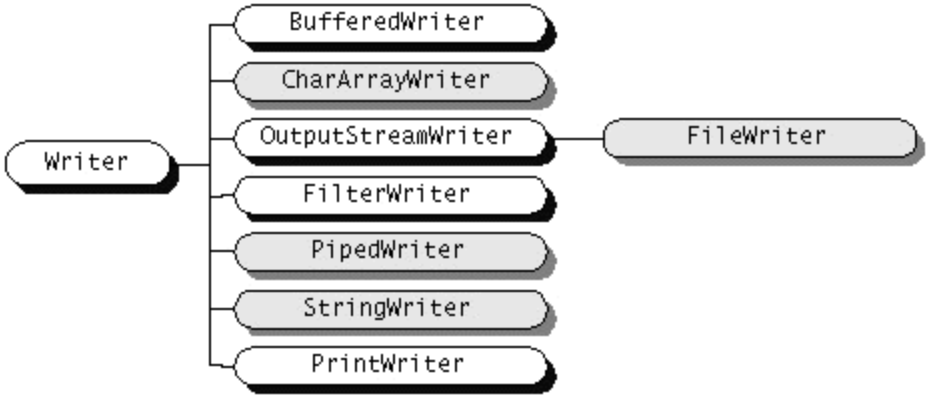
	Wejście	Wyjście
Strumienie bajtowe	**InputStream**	**OutputStream**
Strumienie znakowe	**Reader**	**Writer**

Rodzaje strumieni









Użycie strumieni w operacjach na plikach

1) file << InputStream (np. z jakiejś metody napisanej w Javie)

2) zapis danych binarnych

```
def file = new File('data.bin')
def ints = [1, 2, 3]

file.withDataOutputStream { dos ->
  dos.writeInt(ints.size())
  ints.each { dos.writeInt(it) }
}

def bytes = []
file.eachByte { bytes << it }
println bytes

def sum = 0
file.withDataInputStream { dis ->
  n = dis.readInt()
  n.times {
    sum += dis.readInt()
  }
}
println sum
```

```
[0, 0, 0, 3, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 0, 3]
6
```

3) serializacja (utrwalenie) obiektów

```
def f = new File('o.ser')
def d = new Date()
def list = [1,2,3]

f.withObjectOutputStream {
  it.writeObject(d)
  it.writeObject(list)
}
olist = []
f.eachObject { olist << it }
println olist[0] == d &&
        olist[1] == list
```

true

4) czasem konieczne StringReader i StringWriter

Pliki - zarządzanie zasobami

1) otwarcie pliku

2) operacje

3) zamknięcie pliku (i np. wymiatanie buforów)

Omówione metody (setText(), getText(), each.., with...) biorą to na siebie.

Ponadto nie wymuszają obsługi błędów (wyjątków).

Uwaga na efektywność

Konstrukcje typu getText(), eachLine { }, setText() itp. pod spodem obsługują cały cykl zarządzania zasobami (otwarcie, wykonanie operacji, zamknięcie). **Dotyczy to też metod write i append oraz operatora <<**. Zatem używajmy with dla dużych plików

```
long start
def startTimer = { start = System.currentTimeMillis() }
def elapsed = { System.currentTimeMillis() - start }


def n = 1000
File out = new File('forappend.txt')
out.text = ''
startTimer()
n.times { out << "Linia $it\n" }
println elapsed()
out.text = ''
startTimer()
out.withPrintWriter { writer ->
  n.times {
    writer.println "Linia $it"
  }
}
println elapsed()
```

```
3313
22
```

Strumienie nie-plikowe

Od wielu źrodeł/odbiorników (Socket, URL) możemy pobrać InputStream i OutputStream. I na nich stosować te wszystkie metody (getText, getBytes, readLines, eachLine itp.) jak na plikach.

Jeszcze lepiej – skrót dla klasy URL.

url = new URL(…);

Zamiast pisać

url.inputStream.text

piszemy:

url.text

itp.

Przykład

Strona w internecie (np. jakiegoś hotelu) zawiera, wśród innych tekstów, linki do obrazków. Nazwy interesujących nas obrazków mają formę:

gallery-*nr*.jpg.

Poniższy kod ściąga wszystkie takie obrazki i zapisuje je w podkatalogu 'loaded' bieżącego katalogu.

```
def site = 'http://www.grand*****.com/images/'
def cont = new URL(site).text
def imgsToFind = cont.findAll(/<a href="(gallery-\d+\.jpg)">/) { all, sel ->
  sel
}
def dir = new File('loaded')
imgsToFind.each { fname->
  new File(dir, fname).bytes = new URL(site + '/' + fname).bytes
}
```

Inne metody klasy File

Najważniejsze:

exists() // czy istnieje?
getName() // nazwa
getCanonicalPath() // pełna ścieżka
isDirectory() // czy to katalog
lastModified() // data ostatniej modyfikacji
length() lub size() // rozmiar w bajtach

mkdir(...) // utwórz katalog
mkdirs(...) // utwórz katalog z wszystkimi nadkatalogami
delete() // usuń plik
deleteDir() // usun katalog z całą zawartością
renameTo(...) // zmień nazwę

Uwaga: do metod getProp() bez argumentów można odwoływac się

f.prop

np. canonicalPath

Przykład

```
def fnames = ['c:/ppp/ppp', '.',
              'test1.txt', 'src']

fnames.each { showProps(it) }

def showProps(fname) {
  println "Passed filename is: $fname"
  def f = new File(fname)
  println f.name
  println f.path
  println f.absolutePath
  println f.canonicalPath
  if (f.exists()) {
    println f.isDirectory()
    println f.size()
    println f.lastModified()
  }
  else println "File doesn't exists"
}
```

```
Passed filename is: c:/ppp/ppp
ppp
c:\ppp\ppp
c:\ppp\ppp
C:\ppp\ppp
File doesn't exists
Passed filename is: .
.
.
E:\cdir
true
0
1250325759531
Passed filename is: test1.txt
test1.txt
test1.txt
E:\cdir\test1.txt
E:\cdir\test1.txt
false
76
1250134599468
Passed filename is: src
src
src
E:\cdir\src
E:\cdir\src
true
0
1250328771000
```

Przeglądanie katalogów

def dir = new File(dir)

Metody:

def list = dir.list(...) lub listFile(...) // lista plików w katalogu (Java)

dir.eachFile { closure } // dla każdego pliku (i katalogu) wykonaj

dir.eachDir { closure } // dla każdego katalogu wykonaj

dir.eachFileRecurse { closure } // j.w. wchodząc w podkatalogi

dir.eachDirRecurse { closure }

dir.eachFileMatch(filter) { closure } // o nazwach wg filtra

dir.eachDirMatch(filter) { closure }

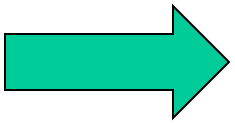
bardzo elastyczne i uniwersalne metody tarverse(), mające charakter "walk-file-tree", ale o dużo większych możliwościach niż w Javie - zob. dokumentację Groovy JDK (GDK) - klasy File lub Path.

Przykład 1

```
def dir = new File('../TestDir')

print '\n--- eachFile --- '
dir.eachFile { show(it)  }
print '\n--- eachDir --- '
dir.eachDir  { show(it)  }
print '\n--- eachFileRec --- '
dir.eachFileRecurse { show(it) }
print '\n--- eachDirRec --- '
dir.eachDirRecurse { show(it) }

def show(f) {
  print '\n' + f.name
  if (f.isDirectory())
    print ' - dir'
}
```



```
--- eachFile ---
Calc1.groovy
Join.groovy
LineSplit.groovy
normalized.txt
platform.txt
subDir1 - dir
subDir2 - dir
--- eachDir ---
subDir1 - dir
subDir2 - dir
--- eachFileRec ---
Calc1.groovy
Join.groovy
LineSplit.groovy
normalized.txt
platform.txt
subDir1 - dir
ClipTest.class
LitZn.class
subDir2 - dir
ClipTest.java
LitZn.java
subDir21 - dir
p.txt
--- eachDirRec ---
subDir1 - dir
subDir2 - dir
subDir21 - dir
```

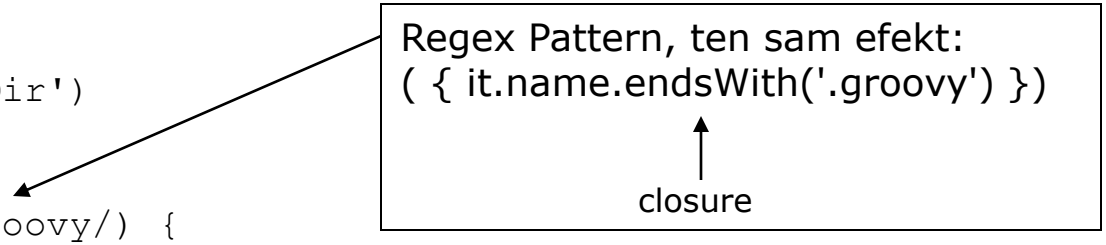
each...Match

eachFileMatch(filter) { closure}

filter - jak klasyfikator w switch (testowana jest nazwa pliku)

```
def dir = new File('../TestDir')

def list = []
dir.eachFileMatch( ~/\w+\.groovy/) {
  if (it.text.contains('[')) list << it.name
}
println list
```



Output:
[Calc1.groovy, Join.groovy, LineSplit.groovy]