Groovy Programming

Arithmetic

Integer arithmetic

Expression	Method call	Result
5 + 3	5.plus(3)	8
5 - 3	5.minus(3)	2
5 * 3	5.multiply(3)	15
5 / 3	5.divide(3)	1.6666666667
5 % 3	5.mod(3)	2

Increment & Decrement

def var1 = 12
var1++ = increment, then run
++var1 = run then increment
var1++ = var1.next()
var1--= var1.previous()

Relational Operators

5<3	5.compareTo(3) < 0	false
5<=3	5.compareTo(3) <= 0	false
5>3	5.compareTo(3) > 0	true
5>=3	5.compareTo(3) >= 0	false
5 == 3	5.equals(3)	false
5 != 3	! 5.equals(3)	true
5 <=>3	5.compareTo(3)	+1

Mixed arithmetic

Expression Method call Result
5 + 3.2 5.plus(3.2) 8.2
5.6 + 3 5.6.plus(3) 8.6
5 - 3.2 5.minus(3.2) 1.8
5.6 - 3 5.6.minus(3) 2.6
5 * 3.2 5.multiply(3.2) 16.0
5.6 * 3 5.6.multiply(3) 16.8
5 / 3.2 5.divide(3.2) 1.5625
5.6 / 3 5.6.divide(3) 1.8666666667

Relational operators

Expression Method call Result
5 < 3 5.compareTo(3) < 0 false
5 <= 3 5.compareTo(3) <= 0 false
5 > 3 5.compareTo(3) > 0 true
5 >= 3 5.compareTo(3) >= 0 true

Equality operators

Expression Method call Result
5 == 3 5.equals(3) false
5 != 3 ! 5.equals(3) // see Appendix C true
5 <=> 3 5.compareTo(3) +1

Some examples are:
def forename = “Ken”
def surname = “Barclay”
forename == “Ken” // true
surname != “Barkley” // true

String literals

Literal	Description
’’	Empty string
’He said “Hello”!’	Single quotes (with nested double quotes)
“He said ’Hello’!”	Double quotes (with nested single quotes)
“””one two three”””	Triple quotes
“””Spread over four lines””	Multi-line text using triple quotes

```
def age=25
’My age is ${age}’      // My age is ${age}
“My age is ${age}”      // My age is 25
“””My age is ${age}””” // My age is 25
“My age is \${age}” // My age is ${age}

def greeting=’Hello world’
greeting[4]             // o index from start
greeting[–1]            // d index from end
greeting[1..2]          // el slice with inclusive range (see Chapter 4)
greeting[1..<3]         // el slice with exclusive range (see Chapter 4)
greeting[4..2]          // oll backward slice
greeting[4, 1, 6]       // oew selective slicing

def greeting=‘Hello world’
’Hello’+‘world’         // Hello world concatenate
‘Hello’ * 3             // HelloHelloHello repeat
greeting– ‘o world’     // Hell remove first occurrence
greeting.size()         // 11 synonymous with length
greeting.length()        // 11 synonymous with size
greeting.count(‘o’)     // 2
greeting.contains(‘ell’) // true
```

TABLE 3.2 String methods

Name	*Signature/description*
`center` *	`String center(Number numberOfChars)` Returns a new `String` of length `numberOfChars` consisting of the recipient padded on the left and right with space characters.
`center` *	`String center(Number numberOfChars, String padding)` Returns a new `String` of length `numberOfChars` consisting of the recipient padded on the left and right with `padding` characters.
`compare-ToIgnoreCase`	`int compareToIgnoreCase(String str)` Compares two strings lexicographically, ignoring case differences.
`concat`	`String concat(String str)` Concatenates the specified `String` to the end of this `String`.
`eachMatch` *	`void eachMatch(String regex, Closure clos)` Processes each `regex` group (see next section) matched substring of the given `String`. The object passed to the closure (see Chapter 9) is an array of strings, following a successful match.
`endsWith`	`Boolean endsWith(String suffix)` Tests whether this string ends with the specified suffix.
`equalsIgnore-Case`	`Boolean equalsIgnoreCase(String str)` Compares this `String` to another `String`, ignoring case considerations.

`getAt` *	`String getAt(int index)`
	`String getAt(IntRange range)`
	`String getAt(Range range)`
	The subscript operator for a `String`.

`indexOf`	`Int indexOf(String str)` Returns the index within this `String` of the first occurrence of the specified substring.
`leftShift` *	`StringBuffer leftShift(Object value)` Overloads the `leftShift` operator to provide an easy way to append multiple objects as `String` representations to a `String`.
`length`	`int length()` Returns the length of the `String`.
`matches`	`Boolean matches(String regex)` Tells whether a `String` matches the given regular expression.
`minus` *	`String minus(Object value)` Remove the `value` part of the `String`.
`next` *	`String next()` This method is called by the ++ operator for the class `String`. It increments the last character in the given `String`.

`padLeft` *	`String padLeft(Number numberOfCharacters)` Pad the `String` with the spaces appended to the left.
`padLeft` *	`String padLeft(Number numberOfCharacters, String padding)` Pad the `String` with the `padding` characters appended to the left.
`padRight` *	`String padRight(Number numberOfCharacters)` Pad the `String` with the spaces appended to the right.
`padRight` *	`String padRight (Number numberOfCharacters, String padding)` Pad the `String` with the `padding` characters appended to the right.
`plus` *	`String plus(Object value)` Appends a `String`.
`previous` *	`String previous()` This method is called by the – operator for the class `String`. It decrements the last character in the given `String`.
`replaceAll`	`void replaceAll(String regex, Closure clos)` Replaces all occurrences of a captured group by the result of a closure on that text.
`reverse` *	`String reverse()` Creates a new `String` which is the reverse of this `String`.
`size` *	`int size()` Returns the length of the `String`.
`split` *	`String[] split(String regex)` Splits this `String` around matches of the given regular expression.
`substring`	`String substring(int beginIndex)` Returns a new `String` that is a substring of this `String`.
`substring`	`String substring(int beginIndex, int endIndex)` Returns a new `String` that is a substring of this `String`.
`toCharacter` *	`Character toCharacter()`
`toDouble` *	`Double toDouble()`
`toFloat` *	`Float toFloat()`
`toInteger` *	`Integer toInteger()`
`toLong` *	`Long toLong()` `String` conversions.
`toList` *	`List toList()` Converts the given `String` into a `List` of strings of one character.
`toLowerCase`	`String toLowerCase()` Converts all of the characters in this `String` to lower case.
`toUpperCase`	`String toUpperCase()` Converts all of the characters in this `String` to upper case.
`tokenize` *	`List tokenize()` Tokenize a String with a space character as delimiter.
`tokenize` *	`List tokenize(String token)` Tokenize a String with `token` as delimiter.

```
'Hello'.compareToIgnoreCase('hello')     // 0
'Hello'.concat('world')                  // Hello world
'Hello'.endsWith('lo')                   // true
'Hello'.equalsIgnoreCase('hello')        // true
'Hello'.indexOf('lo')                    // 3
'Hello world'.indexOf('o', 6)            // 7
'Hello'.matches('Hello')                 // true
'Hello'.matches('He')                    // false
'Hello'.replaceAll('l', 'L')             // HeLLo
'Hello world'.split('l')                 // 'He', 'o wor', 'd'
'Hello'.substring(1)                     // ello
'Hello'.substring(1, 4)                  // ell
'Hello'.toUpperCase()                    // HELLO
def message = 'Hello'
message.center(11)                       //□□□Hello□□□
message.center(3)                        // Hello
message.center(11, '#')                  // ###Hello###
message.eachMatch('.') { ch ->           // print H e l l o
  println ch }                             on separate lines
message.getAt(0)                         // H
message.getAt(0..<3)                     // Hel
message.getAt([0, 2, 4])                 // Hlo
message.leftShift('world')               // Hello world
message << 'world'                       // Hello world
message.minus('ell')                     // Ho
message-'ell'                            // Ho
message.padLeft(4)                       // Hello
message.padLeft(11)                      // □□□□□□Hello
message.padLeft(11, '#')                 // ######Hello
message.padRight(4)                      // Hello
message.padRight(11)                     // Hello□□□□□□
message.padRight(11, '#')                // Hello######
message.plus('world')                    // Hello world
message+'world'                          // Hello world
message.replaceAll('[a-z]') { ch ->      // HELLO
  ch.toUpperCase() }
message.reverse()                        // olleH
message.toList()                         // ['H', 'e', 'l', 'l', 'o']
def message='Hello world'
message.tokenize()                       // ['Hello', 'world']
message.tokenize('l')                    // ['He', 'o wor', 'd']
```

def greeting=‘Hello world’
’Hello’+‘world’ // Hello world concatenate
‘Hello’ * 3 // HelloHelloHello repeat
greeting– ‘o world’ // Hell remove first occurrence
greeting.size() // 11 synonymous with length
greeting.length() // 11 synonymous with size
greeting.count(‘o’) // 2
greeting.contains(‘ell’) // tru

RegEx

def regex = ~’cheese’
’cheesecake’=~’cheese’
!(’cheesecake’=~’fromage’)
’cheesecake’=~regex

=~'regex string'

^	start
$	end
+	one or more times
*	zero or more times
?	zero or once
{}	specific number of instances
[]	character classes
.	wildcard (\\. escaped for decimal)

'aaaaab'=~'a*b'
'b'=~'a*b'
'aaacd'=~'a*c?d'
'aaad'=~'a*c?d'
'aaaaab'=~'a{5}b'
!('aab'=~'a{5}b')

In a regular expression, the period symbol (.) can represent any character. This is described as the wildcard character. Things get complicated, however, when we are required to match an actual period character. All of the following are true:

rhyme=~'.all'
'3.14'=~'3.14' // pattern: 3 followed by any character followed by 14
'3X14'=~'3.14'
'3.14'=~'3\\.14' // pattern: 3.14 literally!
!('3X14'=~'3\\.14')

Great care is required when using the backslash character. In a normal String,it acts as the escape character and so "\\"represents the single backslash. A regular expression to denote a single backslash then becomes "\\\\". Confusingly, we need four occurrences to denote one backslash!

A regular expression may include character classes . A set of characters can be given as a simple sequence of characters enclosed in the metacharacters[and] as in [aeiou]. For letter or number ranges, you can use a dash separator as in [a–z] or [a–mA–M]. The complement of a character class is denoted by a leading caret within the square brackets as in [∧a–z] and represents all characters other than those specified. The value trueis delivered from:

rhyme=~'[HD]umpty'
!(rhyme=~'[hd]umpty')
!(rhyme=~'[∧HD]umpty')

Finally, we can group regular expressions to compose more complex expressions. Groups are formed using the (and) metacharacters. Hence, "(ab)*" is the regular expression for any number of occurrences of ab. You can also use alternation (denoted by │) to match a single regular expression from one of several possible regular expressions. Therefore, "(a│b)*" describes any number of mixed a or b. The following all yield true:

’ababab’==~‘(ab)*’
!(‘ababa’==~‘(ab)*’)
‘ababc’==~‘(ab)*c’
‘aaac’==~‘(a|b)*c’
‘bbbc’==~‘(a|b)*c’
’ababc’==~‘(a |b)*c’

Groovy also permits a pattern to use the / delimiter so that we do not have to double all the backslash symbols. Thus, the example:

def matcher=“\$abc.”=~\\\$(.*)\\.

can also be expressed as:

def matcher=“\$abc.”=~ /\$(.*)\./

Lists, Maps and Ranges

Lists

If the integer index is negative, then it refers to elements by counting from the end. Thus,

numbers [–1] // 14
numbers [–2] // 13
This indexing can also be applied to a Listliteral, as in:
[11, 12, 13, 14][2] // 13

Example	**Description**
[11, 12, 13, 14]	A list of integer values
[‘Ken’, ‘John’, ‘Andrew’]	A list of Strings
[1, 2, [3, 4], 5]	A nested list
[‘Ken’, 21, 1.69]	A heterogeneous list of object references
[ ]	An empty list

List Methods

Name	**Signature/description**
add	boolean add(Object value) Append the new value to the end of this List.
add	void add(int index, Object value) Inserts a new value into this Listat the given index position.
addAll	boolean addAll(Collection values) Append the new valueson to the end of this List.
contains	boolean contains(Object value) Returns true if this List contains the specified value.
flatten *	List flatten() Flattens this Listand returns a new List.
get	Object get(int index) Returns the element at the specified position in this List.
getAt *	Object getAt(int index) Returns the element at the specified position in this List.

getAt *	List getAt(Range range) Return a new Listthat is a sublist of this Listbased on the given range.
getAt *	List getAt(Collection indices) Returns a new Listof the values in this Listat the given indices
intersect *	List intersect(Collection collection) Returns a new Listof all the elements that are common to both the original Listand the input List.
isEmpty	boolean isEmpty() Returns trueif this Listcontains no elements.
leftShift *	Collection leftShift(Object value) Overloads the left shift operator to provide an easy way to append an item to a List.
minus *	List minus(Collection collection) Creates a new Listcomposed of the elements of the original without those specified in the collection.
plus *	List plus(Object value) Creates a new Listcomposed of the elements of the original together with the new value.
plus *	List plus(Collection collection) Creates a new Listcomposed of the elements of the original together with those specified in the collection.
pop *	Object pop() Removes the last item from this List.
putAt *	void putAt(int index, Object value) Supports the subscript operator on the left of an assignment.
remove	Object remove(int index) Removes the element at the specified position in this List.
remove	boolean remove(Object value) Removes the first occurrence in this Listof the specified element.
reverse *	List reverse() Create a new Listthat is the reverse the elements of the original List.
size	int size() Obtains the number of elements in this List.
sort *	List sort() Returns a sorted copy of the original List.

The following shows examples of these Listmethods and their effects.

```
[11, 12, 13, 14].add(15) // [11, 12, 13, 14, 15]
[11, 12, 13, 14].add(2, 15) // [11, 12, 15, 13, 14]
[11, 12, 13, 14].add([15, 16]) // [11, 12, 13, 14, 15, 16]
[11, 12, 13, 14].get(1) // 12
[11, 12, 13, 14].isEmpty() // false
[14, 13, 12, 11].size() // 4
[11, 12, [13, 14]].flatten() // [11, 12, 13, 14]
[11, 12, 13, 14].getAt(1) // 12
[11, 12, 13, 14].getAt(1..2) // [12, 13]
```

[11, 12, 13, 14].getAt([2, 3]) // [13, 14]
[11, 12, 13, 14].intersect([13, 14, 15]) // [13, 14]
[11, 12, 13, 14].pop() // 14
[11, 12, 13, 14].reverse() // [14, 13, 12, 11]
[14, 13, 12, 11].sort() // [11, 12, 13, 14]
Be alert to the following code:
def numbers=[11, 12, 13, 14]
numbers.remove(3)
numbers.remove(13)

Maps

A Map (also known as an associative array, dictionary , table, and hash) is an unordered collection of object references. The elements in a Map collection are accessed by a keyvalue. The keys used in a Map can be of any class. When we insert into a Map collection, two values are required: the key and the value. Indexing the Map with the same key can then retrieve that value. Table 4.3 shows some sample Map literals comprising a comma-separated list of key:value pairs enclosed in square brackets

Example	**Description**
[‘Ken’ : ‘Barclay’, ‘John’ : ‘Savage’]	Forename/surname collection
[4 : [2], 6 : [3, 2], 12 : [6, 4, 3, 2]]	Integer keys and their list of divisors
[:]	Empty map

Observe that if the key in a Map literal is a variable name, then it is interpreted as a Stringvalue. In the example:

def x=1
def y=2
def m=[x : y, y : x]
then, m is the Map:
m=[‘x’ : 2, ‘y’ : 1]

Map methods

Name	**Signature/description**
containsKey	boolean containsKey(Object key) Does this Map contain this key?
get	Object get(Object key) Look up the key in this Map and return the corresponding value. If there is no entry in this Map for the key, then return null.
get *	Object get(Object key, Object defaultValue) Look up the key in this Map and return the corresponding value. If there is no entry in this Map for the key, then return the defaultValue .
getAt *	Object getAt(Object key) Support method for the subscript operator.
keySet	Set keySet() Obtain a Set of the keys in this Map.

put	Object put(Object key, Object value) Associates the specified value with the specified key in this Map. If this Map previously contained a mapping for this key, the old value is replaced by the specified value.
putAt *	Object putAt(Object key, Object value) Support method to allow Maps to operate with subscript assignment.
size	int size() Returns the number of key-value mappings in this Map.
values	Collection values() Returns a collection view of the values contained in this Map.

The following shows examples of these Map methods and their effects.
def mp=[‘Ken’ : 2745, ‘John’ : 2746, ‘Sally’ : 2742]
mp.put(‘Bob’, 2713) // [Bob:2713, Ken:2745, Sally:2742, John:2746]
mp.containsKey(‘Ken’) // true
mp.get(‘David’, 9999) // 9999
mp.get(‘Sally’) // 2742
mp.get(‘Billy’) // null
mp.keySet() // [David, Bob, Ken, Sally, John]
mp.size() // 4
mp[‘Ken’] // 2745

Notice how the valuesmethod returns a Collectionof the values contained in a Map. Often, we find it useful to have these as a List. This is easily achieved with the code: mp.values().asList()

Ranges

A range is shorthand for specifying a sequence of values. A Range is denoted by the first and last values in the sequence, and Range can be inclusive or exclusive. An inclusive Range includes all the values from the first to the last, while an exclusive Range includes all values except the last. Here are some examples of Range literals: 1900..1999 // twentieth century (inclusive Range)

2000..<2100 // twenty-first century (exclusive Range)
‘A’..’D’ // A, B, C, and D
10..1 // 10, 9, ..., 1
‘Z’..’X’ // Z, Y, and X

Observe how an inclusive Range is denoted by .., while an exclusive Range uses ..< between the lower and upper bounds. The range can be denoted by Strings or by Integers. As shown, the Range can be given either in ascending order or in descending order. The first and last values for a Range can also be any numeric integer expression, as in:

def start=10
def finish=20
start..finish+1 // [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]

Range methods

Name	Signature/description
contains	boolean contains(Object obj) Returns trueif this Range contains the specified element.
get	Object get(int index) Returns the element at the specified position in this Range.
getFrom	Comparable getFrom() Get the lower value of this Range.
getTo	Comparable getTo() Get the upper value of this Range.
isReverse	boolean isReverse() Is this a reversed Range, iterating backwards?
size	int size() Returns the number of elements in this Range.
subList	List subList(int fromIndex, int toIndex) Returns a view of the portion of this Range between the specified fromIndex, inclusive, and toIndex, exclusive.

A number of methods are defined to operate with Ranges. They are tabulated in The following show some examples of these Range methods and their effects:

```
def twentiethCentury=1900..1999 // Range literal
def reversedTen=10..1 // Reversed Range
twentiethCentury.size() // 100
twentiethCentury.get(0) // 1900
twentiethCentury.getFrom() // 1900
twentiethCentury.getTo() // 1999
twentiethCentury.contains(2000) // false
twentiethCentury.subList(0, 5) // 1900..1904
reversedTen[2] // 8
reversedTen.isReverse() // true
```

Further details on Lists, Maps, and Ranges are given in Appendix E.

Input/Output

Output

```
print “My name is“
print(“Ken”)
println()
println “My first program”
println(“This is fun”)
```

When we execute this Groovy script, the output produced is:

```
My name is Ken
My first program
This is fun
```

```
def age=25
```

```
print “My age is: “
println age
println “My age is: ${age}”
```

Formatted Output

Formatted output is achieved with the printf method call. The formal description of printfis:

printf(String format, List values)

This method prints its values on the console. The values are any expressions representing what is to be printed. The presentation of these values is under control of the formatting string . This string contains two types of information: ordinary characters, which are simply copied to the output, and conversion specifications, which control conversion and printing of the values.

Two simple illustrations of printf are shown in Example 04. In both examples, the List of values to be printed is empty. The format String comprises ordinary characters. The first printf method call displays its format String.

Further output continues on the same output line. The second example includes the escape character \n, representing a newline. Any output after that will begin on the next line.

```
printf(‘My name is Ken’, [])
printf(‘My name is Ken\n’, [])
```

In the next example, we include a List of values. The format String then includes conversion specifications for each of the values. The conversion specifications are introduced by the percent (%) character. In the first example, %d denotes printing an integer value. In the second, the %f conversion is used for printing floating point values.

```
def a=10
def b=15
printf(‘The sum of %d and %d is %d\n’, [a, b, a+b])
def x=1.234
def y=56.78
printf(‘%f from %f gives %f\n’, [y, x, x– y])
```

The output is:

```
The sum of 10 and 15 is 25
56.780000 from 1.234000 gives –55.546000
```

A more detailed discussion of the conversion specifications is given in Appendix F. Example 06 illustrates using the %s conversion to print a string. In all three examples, the output string is enclosed in [and] to reveal the effect of the conversions. We see that %s simply outputs the string. The conversion %20s outputs the string right justified in a field of 20 characters. The conversion %–20s left justifies the output.

```
printf(‘[%s]\n’, [“Hello there”])
printf(‘[%20s]\n’, [“Hello there”])
printf(‘[%–20s]\n’, [“Hello there”])
```

The output is:

[Hello there]
[Hello there]
[Hello there]

simple input

The object in defined in the Java class System represents the standard input, and is an object of the class InputStream. This class includes the method readLine, which reads a single line of input as a String. Hence, the method call System.in.readLine()can be used to obtain a line of input from the user. Example 07 shows this in a program to read a user's name.

```
print "Please enter your name: "
def name=System.in.readLine()
println "My name is: ${name}"
```

Method readLine returns a String value. We can use the method toInteger on that String to convert it into an integer value or to Double to convert it to a floating point value. Example 08 shows three Groovy methods (see Chapter 7) that might be used to read various input values.

```
def readString() {
        return System.in.readLine()
}
def readInteger() {
        return System.in.readLine().toInteger()
}
def readDouble() {
        return System.in.readLine().toDouble()
}

print 'Please enter your name: '
def name=readString()
println "My name is: ${name}"
print 'Please enter your age: '
def age=readInteger()
println "My age is: ${age}"
```

Using System.in.readLine() implies that each input value is supplied as a single text line. The methods to Integer and to Double expect that the input String be correctly formatted with no unexpected characters and no leading or trailing spaces. The methods readString, readInteger, etc. will find uses in other scripts. We collect them as a set of static methods in the class console that can then be imported into other applications. The class Console is found in the console package. Further information on this class is given in Appendix F.

```
package console
class Console {
        def static readString() { ... }
        def static readLine() { ... }
        def static readInteger() { ... }
        def static readDouble() { ... }
        def static readBoolean() { ... }
```

}

Importantly, the methods of class Console tokenize the input so that, for example, leading whitespace on an integer value is ignored by method readInteger. Example 09 shows how we use these methods.

```
import console.*
print ‘Please enter your name: ‘
def name=Console.readString()
println “My name is: ${name}”
print ‘Please enter your age: ‘
def age=Console.readInteger()
println “My age is: ${age}”
```

Executing this script might result in the following (user input is italicized and bold):

```
Please enter your name: Ken
My name is: Ken
Please enter your age:  25
My age is: 25
```

Importantly, the Console class buffers and tokenizes its input so that more than one value may be given on one input line. Example 10 demonstrates how this is used.

```
import console.*
print ‘Please enter your name and age: ‘
def name=Console.readString()
def age=Console.readInteger()
println “Name: ${name}, and age: ${age}”
```

Running this script, we could have:

```
Please enter your name and age: Ken 225
Name: Ken, and age: 25
```