JN1525-Strings

We can use either single- or double-quotes around strings:

```
assert 'hello, world' == "hello, world"
assert "Hello, Groovy's world" == 'Hello, Groovy\'s world'
    //backslash escapes the quote
assert 'Say "Hello" to the world' == "Say \"Hello\" to the world"
```

Backslashes can escape other characters in Strings. We can use letter codes (eg '\b') or octal codes (eg '\010'):

```
assert '\b' == '\010' //backspace
assert '\t' == '\011' //horizontal tab
assert '\n' == '\012' //linefeed
assert '\f' == '\014' //form feed
assert '\r' == '\015' //carriage return
assert '\\' == '\\' //use backslash to escape the backslash
```

To span multiple lines, use either triple quotes or a backslash at the end of the continuing lines to join them with the next:

```
assert '''hello,
world''' == 'hello,\nworld'
    //triple-quotes for multi-line strings, adds '\n' regardless of host system
assert 'hello, \
world' == 'hello, world' //backslash joins lines within string
```

We can also use three double-quotes.

```
def text = """\
Good morning.
Good night again."""
```

When using double-quotes, either one or three, we can embed code within them using $. Here, they're called GStrings:

```
def name = 'Groovy'
assert "hello $name, how are you today?" == "hello Groovy, how are you today?"
```

Anything more complex than a variable name must be surrounded by curlies:

```
def a = 'How are you?'
assert "The phrase '$a' has length ${a.size()}" ==
    "The phrase 'How are you?' has length 12"
```

We can change the variable's value in the GString:

```
def i= 1, list= []
3.times{ list<< "${i++}" }
assert list.join() == '123'
```

String methods

We can convert other objects in Groovy to their string representation in different ways:

```
def o= new Object()
assert String.valueOf( o ) == o.toString() //this works for any object in Groovy
assert String.valueOf( true ) == true.toString() //boolean value
assert String.valueOf( 'd' as char ) == ('d' as char).toString() //character
assert String.valueOf( 7.5d ) == 7.5d.toString() //double
assert String.valueOf( 8.4f ) == 8.4f.toString() //float
assert String.valueOf( 13i ) == 13i.toString() //integer
assert String.valueOf( 14L ) == 14L.toString() //long
assert String.valueOf( ['a', 'b', 'c'] ) == ['a', 'b', 'c'].toString()
    //list, etc, etc, etc
```

To find the size and substrings:

```
def s= 'abcdefg'
assert s.length() == 7 && s.size() == 7
assert s.substring(2,5) == 'cde' && s.substring(2) == 'cdefg'
assert s.subSequence(2,5) == 'cde'
```

There's different ways to construct a string:

```
assert new String() == ''
assert new String('hello') == 'hello'

def minLowSurr= Character.MIN_LOW_SURROGATE,
    minHighSurr= Character.MIN_HIGH_SURROGATE
def str= 'abc' + minHighSurr + minLowSurr + 'efg'
def ca= ['a', 'b', 'c', minHighSurr, minLowSurr, 'e', 'f', 'g'] as char[]
def ia= ['a', 'b', 'c', 0x10000, 'e', 'f', 'g'] as int[]
assert new String(ca) == str
assert new String(ca, 2, ca.size()-2) == str[2..-1]
assert new String(ia, 2, ia.size()-2) == str[2..-1]

def ca2= new char[8]
str.getChars(0, str.size(), ca2, 0)
    //copy characters from string into character array
assert ca2.size() == str.size()
ca2.eachWithIndex{ elt, i-> assert elt == str[i] }

def ca3= ['a', 'b', 'c', 'd', 'e'] as char[]
'abcde'.toCharArray().eachWithIndex{ it, i-> assert it == ca3[i] }
    //convert String to char array
assert String.valueOf(ca3) == 'abcde' //convert char array to String
assert String.copyValueOf(ca3) == 'abcde' //alternative method name
assert String.valueOf(ca3, 2, 2) == 'cd' //use substring
assert String.copyValueOf(ca3, 2, 2) == 'cd'
```

We can pad and center strings:

```
assert 'hello'.padRight(8,'+').padLeft(10,'+') == '++hello+++'
assert 'hello'.padLeft(7).padRight(10) == '  hello   '
assert 'hello'.center(10, '+').center(14, ' ') == '  ++hello+++  '
```

We can split a string into tokens:

```
assert 'he she\t it'.tokenize() == ['he', 'she', 'it']
    //tokens for split are ' \t\n\r\f'
assert 'he she\t it'.tokenize() ==
    new StringTokenizer('he she\t it').collect{ it }

assert 'he,she;it,;they'.tokenize(',;') == ['he', 'she', 'it', 'they']
    //supply our own tokens
assert new StringTokenizer('he,she;it,;they', ',;').collect{ it } ==
    'he,she;it,;they'.tokenize(',;')

assert new StringTokenizer('he,she,;it', ',;', true).collect{ it } ==
    ['he', ',', 'she', ',', ';', 'it']
  //long form provides extra option to return the tokens with the split-up data
```

Some additional methods:

```
assert 'abcde'.find{ it > 'b' } == 'c' //first one found
assert 'abcde'.findAll{ it > 'b' } == ['c', 'd', 'e'] //all found
assert 'abcde'.findIndexOf{ it > 'c' } == 3 //first one found

assert 'abcde'.every{ it < 'g' } && ! 'abcde'.every{ it < 'c' }
assert 'abcde'.any{ it > 'c' } && ! 'abcde'.any{ it > 'g' }

assert 'morning'.replace('n','t') == 'mortitg' &&
       'boo'.replace('o', 'at') == 'batat' &&
       'book'.replace('oo','ie') == 'biek'

assert 'EggS'.toLowerCase() == 'eggs' && 'EggS'.toUpperCase() == 'EGGS'
assert '   Bacon   '.trim() == 'Bacon'
assert 'noodles'.startsWith('nood') && 'noodles'.endsWith('dles')
assert 'corn soup'.startsWith('rn', 2) //2 is offset

assert 'abc'.concat('def') == 'abcdef'
assert 'abcdefg'.contains('def')
assert ''.isEmpty() && ! 'abc'.isEmpty()

assert 'morning'.indexOf('n') == 3
assert 'morning'.indexOf('n', 4) == 5 //ignore first 4 characters
assert 'morning'.indexOf('ni') == 3
assert 'morning'.indexOf('ni', 4) == -1 //not found
assert 'morning'.lastIndexOf('n') == 5
assert 'morning'.lastIndexOf('n', 4) == 3 //only search first 4 characters
assert 'morning'.lastIndexOf('ni') == 3
assert 'morning'.lastIndexOf('ni', 4) == 3
    //only search first 4 characters for first char of search string
```

We can use operators on strings:

```
assert 'hello, ' + 'balloon' - 'lo' == 'hel, balloon'
    //'-' subtracts one instance at most of string
assert 'hello, balloon' - 'abc' == 'hello, balloon'
assert 'hello, '.plus('balloon').minus('lo') == 'hel, balloon'
    //alternative method syntax
assert 'value is ' + true == 'value is true' &&
       'value is ' + 1.54d == 'value is 1.54' &&
            //first converts double to String (without info loss)
       'value is ' + 7 == 'value is 7' //we can add on various types of values
assert 7 + ' is value' == '7 is value'
assert 'telling true lies' - true == 'telling  lies' &&
       'week has 7 days' - 7 == 'week has  days'
            //we can subtract various types of values
assert 'a' * 3 == 'aaa' && 'a'.multiply(3) == 'aaa'

assert 'hello'.reverse() == 'olleh'
assert 'hello'.count('l') == 2

assert 'abc'.collect{ it * 2 } == ['aa', 'bb', 'cc']
def s= [], t= [:]
'abc'.each{ s << it }
'abc'.eachWithIndex{ elt, i-> t[i]= elt }
assert s == ['a', 'b', 'c'] && t == [0:'a', 1:'b', 2:'c']
assert 'abcde'.toList() == ['a', 'b', 'c', 'd', 'e']

assert 'abc'.next() == 'abd' && 'abc'.previous() == 'abb'
```

We can subscript strings just as we can lists, except of course strings are read-only:

```
assert 'abcdefg'[ 3 ] == 'd'
assert 'abcdefg'.getAt( 3 ) == 'd' //equivalent method name
assert 'abcdefg'.charAt( 3 ) == 'd' //alternative method name
assert 'abcdefg'[ 3..5 ] == 'def'
assert 'abcdefg'.getAt( 3..5 ) == 'def'
assert 'abcdefg'[ 1, 3, 5, 6 ] == 'bdfg'
assert 'abcdefg'[ 1, *3..5 ] == 'bdef'
assert 'abcdefg'[ 1, 3..5 ] == 'bdef'
    //range in subscript flattened automatically
assert 'abcdefg'[-5..-2] == 'cdef'
assert 'abcdefg'.getAt( [ 1, *3..5 ] ) == 'bdef'
assert 'abcdefg'.getAt( [ 1, 3..5 ] ) == 'bdef'

assert 'abcde' == 'ab' + 'c' + 'de'
assert 'abcde'.equals('ab' + 'c' + 'de') //equivalent method name
assert 'abcde'.contentEquals('ab' + 'c' + 'de') //alternative method name
assert 'AbcdE'.equalsIgnoreCase('aBCDe')
assert 'abcde' < 'abcdf' && 'abcde' < 'abcdef'
assert 'abcde'.compareTo('abcdf') == -1 && 'abcde'.compareTo('abcdef') == -1
                                                    //equivalent method
assert 'AbcdEF'.compareToIgnoreCase('aBCDe') == 1
assert 'AbcdE'.compareToIgnoreCase('aBCDef') == -1

assert Collections.max( 'abC'.toList(), String.CASE_INSENSITIVE_ORDER ) == 'C'
assert Collections.min(
            ['abC', 'ABd', 'AbCd'], String.CASE_INSENSITIVE_ORDER ) == 'abC'

assert 'abcde'.regionMatches(2, 'ccccd', 3, 2)
    //match from index 2 in 'abcde' to 2 chars from index 3 in 'ccccd'
assert 'abcDE'.regionMatches(true, 2, 'CCCCd', 3, 2)
    //if first arg is true, ignores case
```

We can format values into a string, using format():

```
//Strings (conversion type 's')
assert String.format('%1$8s', 'hello') == '   hello'
    //width (here, 8) is minimum characters to be written
assert String.format('%2$6s,%1$2s', 'a', 'hello') == ' hello, a'
    //we can re-order arguments
assert String.format('%1$2s', 7, 'd') == ' 7'
    //we can give any type of input; we can ignore arguments
assert String.format('%1s,%2s', null, 'null') == 'null,null'
    //null treated as 'null'
assert String.format('%1$2.4s', 'hello') == 'hell'
    //precision (here, 4) is maximum characters to be written

//Characters ('c')
assert String.format('%1$c,%2$3c', 65, 66 as byte) == 'A,  B'
    //convert argument to character; 2nd value 3 chars wide
assert String.format('%-3c', 67 as short) == 'C  '
    //left-justified with '-' flag; we needn't specify parameter number (1$, etc)
assert String.format('%c', 'D' as char) == 'D'

//Special conversion types:
assert String.format('hello %n world %%') == 'hello \r\n world %'
    //platform-specific newline; double % to quote it

//Boolean ('b')
assert String.format('%b, %b, %b, %b, %b, %b',
                     null, true, false, 0, 1, new Object()) ==
    'false, true, false, true, true, true'
```

StringBuffers

A StringBuffer is a mutable string. (But from Java 5.0 onwards, we should use a StringBuilder instead, because StringBuffers are normally reserved for multi-threaded processing.)

```
def sb1= new StringBuffer(),
    sb2= new StringBuffer('Hello'),
    sb3= new StringBuffer(sb2)
assert sb1.toString() == '' &&
       sb2.toString() == 'Hello' &&
       sb2.toString() == sb3.toString()
```

To find the size and substrings:

```
def sb= new StringBuffer('abcdefg')
assert sb.size() == 7 && sb.length() == 7 //different ways to find size
sb.length= 6 //change size
assert sb.toString() == 'abcdef'
assert sb.reverse().toString() == 'fedcba'
assert sb.toString() == 'fedcba' //reverse() method reverses order permanently
assert sb.substring(2) == 'dcba' //substring from index 2
assert sb.substring(2, 5) == 'dcb' //substring from index 2 to <5
assert sb.subSequence(2, 5) == 'dcb' //substring from index 2 to <5
assert sb + 'zyx' == 'fedcbazyx'
```

To append to a StringBuffer:

```
def sb1= new StringBuffer()
sb1 << 'abc'
sb1 << 'def' << 'ghi' //can chain two << operators
sb1.leftShift('jkl') //equivalent method name
sb1.append('mno') //alternative method name
sb1.append( ['p', 'q', 'r'] as char[] )
sb1.append( ['r', 's', 't', 'u', 'v'] as char[], 1, 3 )
assert sb1.toString() == 'abcdefghijklmnopqrstu'
```

If we append to a String, a StringBuffer is returned:

```
def s= 'foo'
s= s << 'bar'
assert s.class == StringBuffer && s.toString() == 'foobar'
```

As with strings, we can subscript a StringBuffer, returning a string:

```
def sb= new StringBuffer('abcdefg')
assert sb[ 3 ] == 'd'
assert sb[ 3 ].class == String
assert sb.getAt( 3 ) == 'd' //equivalent method name
assert sb.charAt( 3 ) == 'd' //alternative method name
assert sb[ 3..5 ] == 'def'
assert sb[ 1, 3, 5, 6 ] == 'bdfg'
assert sb[ 1, 3..5 ] == 'bdef'
assert sb[-5..-2] == 'cdef'
sb[ 3..5 ] = 'xy' //use subscripts to update StringBuffer
assert sb.toString() == 'abcxyg'
sb.putAt( 2..4, 'z' ) //equivalent method name
assert sb.toString() == 'abzg'
sb.setCharAt(1, 'm' as char) //alternative method name
assert sb.toString() == 'amzg'
```

We can insert into, replace within, and delete from StringBuffers using methods:

```
def sb= new StringBuffer('hello park')
sb.delete(4, 7)
assert sb.toString() == 'hellark'
sb.deleteCharAt(3)
assert sb.toString() == 'helark'
def ca= new char[6]
sb.getChars(2, 5, ca, 1)
    //for indexes 2 to <5, copy into ca beginning from index 1
(['\0', 'l', 'a', 'r', '\0', '\0'] as char[]).
    eachWithIndex{ elt, i-> assert ca[i] == elt }

sb.insert(4, 'se')
assert sb.toString() == 'helaserk'
sb.insert(4, new StringBuffer('ct ') )
assert sb.toString() == 'helact serk'
sb.insert(10, ['i', 'c'] as char[] )
assert sb.toString() == 'helact serick'
sb.insert(6, ['m', 'a', 'l', 't'] as char[], 1, 2)
    //insert 2 chars from subscript 1
assert sb.toString() == 'helactal serick'
sb.insert(10, 'snapla', 3, 5) //insert chars from subscript 3 to <5
assert sb.toString() == 'helactal splerick'
sb.replace(4, 13, 'dor') //replace chars from subscript 4 to <13
assert sb.toString() == 'heladorrick'
```

We can find the index of substrings:

```
def sb= new StringBuffer('hello elm')
assert sb.indexOf('el') == 1
assert sb.indexOf('el', 3) == 6 //first occurence of 'el' from index 3
assert sb.lastIndexOf('el') == 6
assert sb.lastIndexOf('el', 3) == 1 //last occurence of 'el' up to index 3
```

Some miscellaneous methods:

```
def s= new String( new StringBuffer('abcdefg') )
assert s == 'abcdefg'
assert s.contains('def')
assert s.contentEquals('abcdefg')
assert s.contentEquals( new StringBuffer('abcdefg') )
def s2= s.replace('def', 'xyz')
assert s2 == 'abcxyzg'
```

We can enquire of code points in a String or StringBuffer using methods on them, just as we can with methods on Character:

```
def minLowSurr= Character.MIN_LOW_SURROGATE,
    minHighSurr= Character.MIN_HIGH_SURROGATE

def s1= 'abc'+ minHighSurr + minLowSurr +'efg'
assert s1.codePointAt(3) == 0x10000 //if high surrogate, add on low surrogate
assert s1.codePointAt(4) == minLowSurr //if low surrogate, use it only
assert s1.codePointAt(5) == 'e' as int
assert s1.codePointBefore(4) == minHighSurr
assert s1.codePointBefore(5) == 0x10000
    //if low surrogate, look back more for high one, and use both
assert s1.codePointCount(1, 5) == 3
    //number of code points in a substring with indexes >=1 and <5
assert s1.offsetByCodePoints(1, 3) == 5
    //index from 1 that's offset by 3 code points

def sb= new StringBuffer( 'abc'+ minHighSurr + minLowSurr +'efg' )
    //also, for StringBuffers
assert sb.codePointAt(5) == 'e' as int
assert sb.codePointBefore(4) == minHighSurr
assert sb.codePointCount(1, 5) == 3
assert sb.offsetByCodePoints(1, 3) == 5

sb.appendCodePoint(0x10000)
assert sb.toString() ==
    'abc'+ minHighSurr + minLowSurr +'efg'+ minHighSurr + minLowSurr
```

We can manipulate the implementation of a StringBuffer:

```
def sb1= new StringBuffer() //default initial capacity is 16
assert sb1.capacity() == 16

def sb2= new StringBuffer(5) //we can specify initial capacity
assert sb2.capacity() == 5
sb2<< 'abc'
assert sb2.capacity() == 5 && sb2.size() == 3
sb2.trimToSize()
assert sb2.capacity() == 3
sb2.ensureCapacity(10)
assert sb2.capacity() == 10

def sb3= new StringBuffer(0) //capacity approximately doubles when required
def cap= 0, caps=[]
100.times{
  if((sb3<< 'a').capacity() != cap) caps<< (cap= sb3.capacity())
}
assert caps == [2, 6, 14, 30, 62, 126]
```