CHAPTER 4

LISTS, MAPS, AND RANGES

In this chapter, we introduce the *list*, *map*, and *range*. All are *collections* of references to other objects. The `List` and `Map` can reference objects of differing types. The `Range` represents a collection of integer values. The `List` and `Map` also grow dynamically. Each object referenced in a `List` is identified by an integer *index*. In contrast, a `Map` collection is indexed by a value of any kind. Because the class of the objects maintained by these collections is arbitrary, the elements of a `List` might be a `Map` and the elements of a `Map` might be a `List`. In this way, we can create data structures of arbitrary complexity. This is examined in Chapter 6 and in subsequent chapters.

4.1 LISTS

The `List` is a structure used to store a collection of data items. In Groovy, the `List` holds a sequence of object references. Object references in a `List` occupy a position in the sequence and are distinguished by an integer index. A `List` literal is presented as a series of objects separated by commas and enclosed in square brackets. Table 4.1 illustrates sample `List` literals.

To process the data in a list, we must be able to access individual elements. Groovy `Lists` are indexed using the indexing operator `[]`. `List` indices start at zero, which refers to the first element. Consider the following `List` object identified as `numbers` and some sample `List` accessing.

```
def numbers = [11, 12, 13, 14]    // list with four items
numbers [0]                       // 11
numbers [3]                       // 14
```

TABLE 4.1 List literals

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

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 `List` literal, as in:

```
[11, 12, 13, 14][2]   // 13
```

Once again, it is worth restating that the [] operator is the method `getAt` (see Section 4.2) defined in the `List` class. Hence, in addition to referring to a list element as `numbers[3]`, we need to recognize that, in fact, we are invoking the `getAt` method on the `List` object `numbers` with the method parameter 3, as in `numbers.getAt(3)`.

Additionally, we can index a `List` using ranges (examined later in this chapter). An *inclusive range* of the form `start..end` delivers a new `List` object comprising the references to the objects from the original `List` starting at index position `start` and ending at index position `end`. An *exclusive range* of the form `start..<end` includes all elements except the final `end` element. Examples of range indices are:

```
numbers [0..2]   // [11, 12, 13]
numbers [1..<3]  // [12, 13]
```

The `List` indexing operator can also be used to set new values into a `List`. Used on the left side of an assignment, the element at the given position is replaced by the value on the right of the assignment. The index can only be a single integer expression. If the replacement value on the right side of the assignment is itself a `List`, then it is used as the replacement.

```
numbers [1] = 22          // [11, 22, 13, 14]
numbers [1] = [33, 44]    // [11, [33, 44], 13, 14]
```

This assignment is provided by the method putAt (see Table 4.2).

A new item can be appended on to the right end of a List using the << operator (the leftShift method) as in:

```
numbers << 15      // [11, [33, 44], 13, 14, 15]
```

Equally, the + operator (the plus method) is used to concatenate Lists:

```
numbers = [11, 12, 13, 14]     // list with four items
numbers + [15, 16]             // [11, 12, 13, 14, 15, 16]
```

The – operator (the minus method) is used to remove items from a List:

```
numbers = [11, 12, 13, 14]     // list with four items
numbers - [13]                 // [11, 12, 14]
```

4.2 LIST METHODS

The Groovy List class supports a host of methods that make list processing pleasing and easy. The List class removes much of the work that would otherwise have to be programmed in an application. Table 4.2 tabulates and describes some of the more common List methods. Note that those methods with an asterisk are the augmented GDK methods.

TABLE 4.2 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 `List` at the given `index` position.
addAll	`boolean addAll(Collection values)` Append the new `values` on 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 `List` and returns a new `List`.
get	`Object get(int index)` Returns the element at the specified position in this `List`.

Continued

TABLE 4.2 List methods (*Continued*)

Name	*Signature/description*
`getAt` *	`Object getAt(int index)` Returns the element at the specified position in this `List`.
`getAt` *	`List getAt(Range range)` Return a new `List` that is a sublist of this `List` based on the given `range`.
`getAt` *	`List getAt(Collection indices)` Returns a new `List` of the values in this `List` at the given `indices`
`intersect` *	`List intersect(Collection collection)` Returns a new `List` of all the elements that are common to both the original `List` and the input `List`.
`isEmpty`	`boolean isEmpty()` Returns `true` if this `List` contains 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 `List` composed of the elements of the original without those specified in the `collection`.
`plus` *	`List plus(Object value)` Creates a new `List` composed of the elements of the original together with the new `value`.
`plus` *	`List plus(Collection collection)` Creates a new `List` composed 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 `List` of the specified element.
`reverse` *	`List reverse()` Create a new `List` that 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 `List` methods 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)
```

The first `remove` call seeks to remove the item at position 3 using the method `remove(int index)`. The statement has the desired effect. However, in the second call, the programmer intends to remove the value 13 using the method `remove(Object value)`. This fails, reporting an out-of-bounds exception, having attempted to call the first `remove` method. Had the `List` contained, say, `String` values, as in the following, then everything operates as expected. Groovy is able to correctly identify the required calls to `remove`.

```
def names=['Ken', 'John', 'Sally', 'Jon']
names.remove(3)
names.remove('Ken')
```

4.3 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 *key* value. 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.

TABLE 4.3 Map literals

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 `String` value. 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]
```

Individual elements of a `Map` are accessed using the subscript operator (implemented by the method `getAt`; see Section 4.4). This time, the index value can be any class of object and represents a key. The value returned is the value paired with the key or the value `null` if no entry with that key exists. Consider the following `Map` objects referenced as `names` and `divisors`, and some simple indexing:

```
def names=['Ken' : 'Barclay', 'John' : 'Savage']
def divisors=[4 : [2], 6 : [3, 2], 12 : [6, 4, 3, 2]]
names['Ken']                                          // 'Barclay'
names.Ken                                             // 'Barclay'
names['Jessie']                                       // null
divisors[6]                                           // [3, 2]
```

As with `Lists`, this indexing is provided by the `getAt` method (see next section). Equally, the `putAt` method supports indexing on the left of an assignment, as in:

```
divisors[6]=[6, 3, 2, 1]          // [4 : [2], 6 : [6, 3, 2, 1],
                                  // 12 : [6, 4, 3, 2]]
```

We must be especially careful to recognize that the keys for `Maps` are objects. In the `names Map` given previously, the keys are `String` objects. Similarly, the keys of the `divisors` map are `Integer` objects. Hence, it is perfectly possible to have the `Map`:

```
def careful=[ 1 : 'Ken', '1' : 'Barclay']
careful[1]                                  // Ken
careful['1']                                // Barclay
```

The first entry has the `Integer` key assigned as `1`, while the second entry has the `String` key assigned as `'1'`. We might have arrived at this by receiving input from the user, reading the first key as an `Integer` and the second key as a `String`. We must be especially diligent if this was not our intention.

4.4 MAP METHODS

The `Map` class supports a range of methods that make map processing very easy. The `Map` class removes much of the work that would otherwise have to be programmed in an application where a set of relationships need to be formed between pairs of objects. Table 4.4 tabulates and describes some of the more common `Map` methods. Again, those marked with an asterisk are courtesy of the GDK.

TABLE 4.4 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 values method returns a Collection of 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()
```

4.5 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]
```

TABLE 4.5 Range methods

Name	*Signature/description*
`contains`	`boolean contains(Object obj)` Returns `true` if 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 Table 4.5.

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.

4.6 EXERCISES

1. Given the list `[14, 12, 13, 11]`, express how we would obtain the `List` with these elements in descending order.
2. Determine the effect of the expression `[1, [2, [3, 4]]].flatten()`, and then state whether `flatten` recurses through nested `Lists`.