Kotlin Collections and Collection Extension Functions Cheat Sheet

Creating Collections

Arrays		
Simple Array	`val intArray: Array< Int> = arrayOf(1, 2, 3)`	
Simple Array of Primitives	`val primitiveIntArray: IntArray = intArrayOf(1, 2, 3)`	Or `doubleArrayOf(1,2,3)` / `longArrayOf(1,2,3)` / `floatArrayOf(1,2,3)` etc.
Copy of Array	`val copyOf Array: Array< Int> = intArr ay.c op yOf()`	
Partial copy of Array	`val partia lCo pyO fArray: Array< Int> = intArr ay.c op yOf Ran ge(0, 2)`	
Lists		
Simple List	`val intList: List<I nt> = listOf(1, 2, 3)`	Or `array Lis tOf (1, 2,3)`
Empty List	`val emptyList: List<I nt> = emptyL ist()`	Or `listOf()`
List with no null elements	`val listWi thN onN ull Ele ments: List<I nt> = listOf Not Null(1, null, 3)`	same as `List( 1,3)`
Sets		
Simple Set	`val aSet: Set<In t> = setOf(1)`	Or `hashS etO f(1)` / `linke dSe rOf(1)`
Empty Set	`val emptySet: Set<In t> = emptyS et()`	Or `setOf()` / `hashS etOf()` / `linke - dSe tOf()`
Maps		
Simple Map	`val aMap: Map<St ring, Int> = mapOf( " hi" to 1, " hel lo" to 2)`	Or `mapOf (Pa ir( " hi", 1)` / `hashM - apO f("h i" to 1)` / `linke dMa pOf - ("hi " to 1)`
Empty Map	`val emptyMap: Map<St ring, Int> = emptyM ap()`	Or `mapOf()` / `hashM apOf()` / `linke - dMa pOf()`
Black sheep, mutables		
Simple Mutable List	`val mutabl eList: Mutabl eLi st< Int> = mutabl eLi stOf(1, 2, 3)`	
Simple Mutable Set	`val mutabl eSet: Mutabl eSe t<I nt> = mutabl eSe tOf(1)`	
Simple Mutable Map	`var mutabl eMap: Mutabl eMa p<S tring, Int> = mutabl eMa pOf ("hi " to 1, " hel lo" to 2)`	

We will be using these collections throughout the cheat sheet.

Operators

Method	Example	Result	Explanation
		Iterables	
Plus	`intList + 1`	`[1, 2, 3, 1]`	Returns a new iterables with old values + added one
Plus (Iterable)	`intList + listOf( 1, 2, 3)`	`[1, 2, 3, 1, 2, 3]`	Returns a new iterable with old values + values from added iterable
Minus	`intList - 1`	`[2, 3]`	Returns a new iterable with old values - subtracted one
Minus (Iterable)	`intList - listOf(1, 2)`	`[3]`	Returns a new iterable with old values without the values from subtracted iterable
		Maps	
Plus	`aMap + Pair("H i", 2)`	`{hi=1, hello=2, Goodby e=3}`	Returns new map with old map values + new Pair. Updates value if it differs
Plus (Map)	`aMap + mapOf( Pai r("h ell o", 2), Pair("G ood bye ", 3))`	`{hi=1, hello=2, Goodby e=3}`	Returns new map with old map values + Pairs from added map. Updates values if they differ.
Minus	`aMap - Pair("H i", 2)`	`{Hi=2}`	Takes in a key and removes if found
Minus (Map)	`aMap - listOf ("he llo ", " hi")`	`{}`	Takes in an iterable of keys and removes if found
		Mutables	
Minus Assign	`mutab leList -= 2`	`[1, 3]`	Mutates the list, removes element if found. Returns boolean
Plus Assign	`mutab leList += 2`	`[1, 3, 2]`	Mutates the list, adds element. Returns boolean
Minus Assign (MutableMap)	`mutab leM ap.m in usA ssi gn( " hel lo")`	`{hi=1}`	Takes in key and removes if that is found from the mutated map. Returns boolean. Same as `-=`
Plus Assign (MutableMap)	`mutab leM ap.p lu sAs sig n("G ood bye " to 3)`	`{hi=1, Goodbye=3}`	Takes in key and adds a new pair into the mutated map. Returns boolean. Same as `+=`

Transformers

Method	Example	Result	Explanation
Associate	`intLi st.a ss ociate { Pair(i t.t oSt ring(), it) }`	`{1=1, 2=2, 3=3}`	Returns a Map containing key-value pairs created by lambda
Map	`intLi st.map { it + 1 }`	`[2,3,4]`	Returns a new list by transforming all elements from the initial Iterable.
MapNotNull	`intLi st.m ap NotNull { null }`	`[]`	Returned list contains only elements that return as not null from the lamdba
MapIndexed	`intLi st.m ap Indexed{ idx, value -> if (idx == 0) value + 1 else value + 2 }`	`[2,4,5]`	Returns a new list by transforming all elements from the initial Iterable. Lambda receives an index as first value, element itself as second.
MapIndexedNotNull	`intLi st.m ap Ind exe dNo tNull { idx, value -> if (idx == 0) null else value + 2 }`	`[4,5]`	Combination of Map, MapIndexed & MapIndexedNotNull
MapKeys	`aMap.m apKeys { pair -> pair.key + ", mate" }`	`{hi, mate=1, hello, mate=2}`	Transforms all elements from a map. Receives a Pair to lambda, lamdba return value is the new key of original value
MapValues	`aMap.m ap Values { pair -> pair.value + 2 })`	`{hi=3, hello=4}`	Transforms all elements from a map. Receives a Pair to lambda, lamdba return value is the new value for the original key.
Reversed	`intLi st.r ev ers ed())`	`[3,2,1]`	
Partition	`intLi st.p ar tition { it > 2 })`	`Pair( [3], [1,2])`	Splits collection into to based on predicate
Slice	`intLi st.s li ce( 1..2))`	`[2,3]`	Takes a range from collection based on indexes
Sorted	`intLi st.s or ted())`	`[1,2,3]`	
SortedByDesc-ending	`intLi st.s or ted ByD esc ending { it }`	`[3,2,1]`	Sorts descending based on what lambda returns. Lamdba receives the value itself.
SortedWith	`intLi st.s or ted Wit h(C omp ara tor <In - t> { x, y -> when { x == 2 -> 1 y == 2 -> -1 else -> y - x } })`	`[3,1,2]`	Takes in a Comparator and uses that to sort elements in Iterable.
Flatten	`listO f(i ntList, aSet).f la tten()`	`[2,3, 4,1]`	Takes elements of all passed in collections and returns a collection with all those elements
FlatMap with just return	`listO f(i ntList, aSet).f latMap { it }`	`[2,3, 4,1]`	Used for Iterable of Iterables and Lambdas that return Iterables. Transforms elements and flattens them after transformation.
FlatMap with transform	`listOf (in tList, aSet).f latMap { iterable: Iterab le< Int> -> iterable.map { it + 1 } }`	`[2,3, 4,2]`	FlatMap is often used with monadic containers to fluently handle context, errors and side effects.
Zip	`listOf(3, 4).zip (in tList)`	`[(3,1), (4,2)]`	Creates a list of Pairs from two Iterables. As many pairs as values in shorter of the original Iterables.
Zip with predicate	`listOf(3, 4).zip (in tList) { firstElem, secondElem -> Pair(firstElem - 2, secondElem + 2) }`	`[(1,3), (2,4)]`	Creates a list of Pairs from two Iterables. As many pairs as values in shorter of the original Iterables. Lambda receives both items on that index from Iterables.
Unzip	`listO f(P air ("hi ", 1), Pair("h ell o", 2)).un zip()`	`Pair([hi, hello], [1,2])`	Reverses the operation from `zip`. Takes in an Iterable of Pairs and returns them as a Pair of Lists.

Aggregators

Method	Example	Result	Explanation
		Folds And Reduces	
Fold	`intLi st.f ol d(10) { accumu lator, value -> accumu lator + value }`	`16` *(10+1+2+3)*	Accumulates values starting with initial and applying operation from left to right. Lambda receives accumulated value and current value.
FoldIndexed	`intLi st.f ol dIn dex ed(10) { idx, accumu lator, value -> if (idx == 2) accumu lator else accumu lator + value }`	`13` *(10+1+2)*	Accumulates values starting with initial and applying operation from left to right. Lambda receives index as the first value.
FoldRight	`intLi st.f ol dRi ght(10) { value, accumulator -> accumu lator + value }`	`16` *(10+3+2+1)*	Accumulates values starting with initial and applying operation from right to left. Lambda receives accumulated value and current value.
FoldRightIndexed	`intLi st.f ol dRi ght Ind exe d(10) { idx, value, accumulator -> if (idx == 2) accumu lator else accumu lator + value }`	`16` *(10+3+2+1)*	
Reduce	`intLi st.r educe { accumu lator, value -> accumu lator + value }`	`6` *(1+2+3)*	Accumulates values starting with first value and applying operation from left to right. Lambda receives accumulated value and current value.
ReduceRight	`intLi st.r ed uce Right { value, accumulator -> accumu lator + value }`	`6` *(3+2+1)*	Accumulates values starting with first value and applying operation from right to left. Lambda receives accumulated value and current value.

ReduceIndexed	`intLi st.r ed uce Indexed { idx, accumu - lator, value ->`
	`  if (idx == 2) accumu lator else accumu lator + value`
	`}`
		`3` *(1+2)*	

ReduceRightI-ndexed	`intLi st.r ed uce Rig htI ndexed { idx, value, accumulator ->`
	`  if (idx == 2) accumu lator else accumu lator + value`
	`}`
		`3` *(2+1)*	

		Grouping	
GroupBy	`intLi st.g roupBy { value -> 2 }`	`{2=[1, 2, 3]}`	Uses value returned from lamdba to group elements of the Iterable. All values whose lambda returns same key will be grouped.
GroupBy (With new values)	`intLi st.g ro upBy({ it }, { it + 1 })`	`{1=[2], 2=[3], 3=[4]}`	Same as group by plus takes another lambda that can be used to transform the current value

GroupByTo	`val mutabl eSt rin gTo ListMap = mapOf( " - fir st" to 1,`
	`  " sec ond " to 2)`
	`mutableStringToListMap.values.groupByTo( mutableMapOf<Int, Mutabl eLi st< Int >>(),`
	`{`
	`  value: Int -> value }, { value -> value + 10 })`
		{1=[11], 2=[12]}	Group by first lambda, modify value with second lambda, dump the values to given mutable map

GroupingBy -> FoldTo	`intLi st.g ro upingBy { it }`
	`  .foldTo(mutableMapOf<Int, Int>(), 0) {`
	`  accumu lator, element ->`
	`  accumu lator + element`
	`}`
		`{1=1, 2=2, 3=3}`	Create a grouping by a lambda, fold using passed in lambda and given initial value, insert into given mutable destination object

Grouping > Aggregate	`intLi st.g ro upingBy { " key " }`
	`  .aggregate({`
	`  key, accumu lator: String?,`
	`  element, isFirst ->`
	`  when (accum ulator) {`
	`  null -> " $el eme nt"`
	`  else -> accumu lator + " $el - eme nt"`
	`  }`
	`})`
		`{key= 123}`	Create a grouping by a lambda, aggregate each group. Lambda receives all keys, nullable accumulator and the element plus a flag if value is the first on from this group. If isFirst --> accumulator is null.

Chunked	`val list = listOf("one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten")`
	
	`list.chunked(3)`
		`[[one, two, three], [four, five, six], [seven, eight, nine], [ten]]`	Splits this collection into a list of lists each not exceeding the given size
			The last list in the resulting list may have less elements than the given size.

		Aggregating	
Count	`intLi st.c ou nt()`	`3`	AKA size
Count (with Lambda)	`intLi st.c ount { it == 2 })`	`1`	Count of elements satisfying the predicate
Average	`intLi st.a ve rage()`	`2.0` *((1+2+3)/3 = 2.0)*	Only for numeric Iterables
Max	`intLi st.m ax()`	`3`	Maximum value in the list. Only for Iterables of Compar-ables.
MaxBy	`intLi st.m axBy { it * 3 }`	`3`	Maximum value returned from lambda. Only for Lambdas returning Comparables.
MaxWith	`intLi st.m ax Wit h(o neO rLa rger)`	`1`	Maximum value defined by passed in Comparator
Min	`intLi st.m in()`	`1`	Minimum value in the list. Only for Iterables of Compar-ables.
MinBy	`intLi st.m inBy { it * 3 }`	`1`	Minimum value returned from lambda. Only for Lambdas returning Comparables.
MinWith	`intLi st.m in Wit h(o neO rLa rger)`	`3`	Minimum value defined by passed in Comparator
Sum	`intLi st.s um()`	`6`	Summation of all values in Iterable. Only numeric Iterables.
SumBy	`intLi st.s umBy { if(it == 3) 6 else it })`	`9` *(1+2+6)*	Summation of values returned by passed in lambda. Only for lambdas returning numeric values.
SumByDouble	`intLi st.s um ByD ouble { it.toD ouble() }`	`6.0`	Summation to Double values. Lambdareceives the value and returns a Double.

```
val oneOrLarger = Comparator<Int> { x, y ->
  when{
    x == 1 -> 1
    y == 1 -> -1
    else -> y - x
  }
}
```

Filtering and other predicates + getting individual elements

Method	Example	Result	Notes
		Filtering	
Filter	`intLi st.f ilter { it > 2 }`	`[3]`	Filter-in
FilterKeys	`aMap.f il terKeys { it != " hel lo" }`	`{hi=1}`	
FilterValues	`aMap.f il ter Values { it == 2 }`	`{hell o=2}`	

FilterIndexed	`intLi st.f il ter Indexed { idx, value -> idx == 2 || value == 2 }`	`[2,3]`	
FilterIsInstance	`intLi st.f il ter IsI nst anc e<S tri ng>()`	`[]`	Type parameter defines the class instance. None returned because in our list all of them are ints
	Taking and Dropping		
Take	`intLi st.t ak e(2)`	`[1,2]`	Take n elements from Iterable. If passed in number larger than list, full list is returned.
TakeWhile	`intLi st.t ak eWhile { it < 3 }`	`[1,2]`	
TakeLast	`intLi st.t ak eLa st(2)`	`[2,3]`	
TakeLastWhile	`intLi st.t ak eLa stWhile { it < 3 }`	`[]`	Last element already satisfies this condition --> empty
Drop	`intLi st.d ro p(2)`	`[3]`	Drop n elements from the start of the Iterable.
DropWhile	`intLi st.d ro pWhile { it < 3 }`	`[3]`	
DropLast	`intLi st.d ro pLa st(2)`	`[1]`	
DropLastWhile	`intLi st.d ro pLa stWhile { it > 2 }`	`[1, 2]`	
	Retrieving individual elements		
Component	`intLi st.c om pon ent1()`	`1`	There are 5 of these --> `compo nen t1()compo nen t2()compo nen t3()compo nen t4()compo nen t5()`
ElementAt	`intLi st.e le men tAt(2)`	`3`	Retrieve element at his index. Throws IndexOutOfBounds if element index doesn't exist
ElementAtOrElse	`intLi st.e le men tAt OrE lse(13) { 4 }`	`4`	Retrieve element at his index or return lambda value if element index doesn't exist.
ElementAtOrNull	`intLi st.e le men tAt OrN ull (666)`	`null`	Retrieve element at his index or return null if element index doesn't exist.
Get (clumsy syntax)	`intLi st.g et(2)`	`3`	Get element by index
Get	`intLi st[2]`	`3`	Shorthand and preferred way for the one above
GetOrElse	`intLis t.g etO rEl se(14) { 42 }`	`42`	Get element or return lambda value if it doesn't exist.
Get from Map (clumsy syntax)	`aMap.g et ("hi ")`	`1`	
Get from Map	`aMap[ " hi"]`	`1`	
GetValue	`aMap.g et Val ue( " hi")1`	`1`	Get value or throw NoSuchElementException
GetOrDefault	`aMap.g et OrD efa ult ("HI ", 4)`	`4`	Get value or return the value returned from lambda
GetOrPut	`mutab leM ap.g et OrP ut( " HI") { 5 }`	`5`	MutableMap only. Returns the the value if it exist, otherwise puts it and returns put value.
	Finding		
BinarySearch	`intLi st.b in ary Sea rch(2)`	`1`	Does a binary search through the collection and returns the index of the element if found. Otherwise returns negative index.
Find	`intLi st.find { it > 1 }`	`2`	First element satisfying the condition or null if not found
FindLast	`intLi st.f in dLast { it > 1 }`	`3`	Last element satisfying the condition or null if not found
First	`intLi st.f ir st()`	`1`	First element of Iterable or throws NoSuchElementException
First with predicate	`intLi st.f irst { it > 1 }`	`2`	Same as find but throws NoSuchElementException if not found
FirstOrNull	`intLi st.f ir stO rNu ll()`	`1`	Throw safe version of `first()`.
FirstOrNull with predicate	`intLi st.f ir stO rNull { it > 1 }`	`2`	Throw safe version of `first(() -> Boolean)`.
IndexOf	`intLi st.i nd exO f(1)`	`0`	
IndexOfFirst	`intLi st.i nd exO fFirst { it > 1 }`	`1`	
IndexOfLast	`intLi st.i nd exO fLast { it > 1 }`	`2`	
Last	`intLi st.l ast()`	`3`	Throws NoSuchElementException if empty Iterable
Last with predicate	`intLi st.last { it > 1 }`	`3`	Throws NoSuchElementException if none found satisfying the condition.
LastIndexOf	`intLi st.l as tIn dex Of(2)`	`1`	
LastOrNull	`intLi st.l as tOr Null()`	`3`	Throw safe version of `last()`
LastOrNull with predicate	`intLi st.l as tOrNull { it > 1 }`	`3`	Throw safe version of `last(() -> Boolean)`.
	Unions, distincts, intersections etc.		
Distinct	`intLi st.d is tin ct()`	`[1, 2, 3]`	
DistinctBy	`intLi st.d is tinctBy { if (it > 1) it else 2 }`	`[1,3]`	
Intersect	`intLi st.i nt ers ect (li stOf(1, 2))`	`[1,2]`	
MinusElement	`intLi st.m in usE lem ent(2)`	`[1,3]`	
MinusElement with collection	`intLi st.m in usE lem ent (li stOf(1, 2))`	`[3]`	

Single	`listO f("One Elemen t").s in gle()`	`One Element`	Returns only element or throws.
SingleOrNull	`intLi st.s in gle OrN ull()`	null	Throw safe version of `singl e()`
OrEmpty	`intLi st.o rE mpty()`	`[1, 2, 3]`	Returns itself or an empty list if itself is null.
Union	`intLi st.u ni on( lis tOf (4, 5,6))`	`[1,2, 3,4 ,5,6]`	
Union (infix notation)	`intList union listOf (4, 5,6)`	`[1,2, 3,4 ,5,6]`	

Checks and Actions

Method	Example	Result	Notes
	Acting on list elements		
	`val listOf Fun ctions = listOf({ print( " first ") }, { print( " second ") })`		
ForEach	`listO fFu nct ion s.f orEach { it() }`	`first second`	

ForEachIndexed	`listO fFu nct ion s.f orE ach Indexed { idx, fn ->`
	`  if (idx == 0) fn() else print( " Won't do it")`
	`}`
		`first Won't do it`	

OnEach	`intLi st.o nEach { print(it) }`	`123`	
	Checks		
All	`intLi st.all { it < 4 }`	`true`	All of them are less than 4
Any	`intLi st.a ny()`	`true`	Collection has elements
Any with predicate	`intLi st.any { it > 4 }`	`false`	None of them are more than 4
Contains (standard)	`intLi st.c on tai ns(3)`	`true`	
Contains (idiomatic Kotlin)	`3 in intList`	`true`	
ContainsAll	`intLi st.c on tai nsA ll( lis tOf(2, 3, 4))`	`false`	
Contains (Map, standard)	`aMap.c on tai ns( " Hel lo")`	`false`	Same as `conta ins Key()`
Contains (Map, Idiomatic Kotlin)	`"Hello" in aMap`	`false`	Same as `conta ins Key()`
ContainsKey	`aMap.c on tai nsK ey( " hel lo")`	`true`	Same as `conta ins()`
ContainsValue	`aMap.c on tai nsV alu e(2)`	`true`	
None	`intLi st.n one()`	`false`	There are elements on the list
None with predicate	`intLi st.none { it > 5 }`	`true`	None of them are larger than 5
IsEmpty	`intLi st.i sE mpty()`	`false`	
IsNotEmpty	`intLi st.i sN otE mpty()`	`true`	

<3 Kotlin

Github repository with all code examples:
https://github.com/Xantier/Kollections
Contributions Welcome!

PDF of this cheat sheet:
Download

Created with <3 by Jussi Hallila

Originally created with the help of Cheatography.