Cheatography

Kotlin Collection Extensions Cheat Sheet

by Xantier via cheatography.com/38756/cs/12075/

Creating Collections

Arrays		
Simple Array	`val intArray: Array<Int> = arrayOf(1, 2, 3)`	
Copy of Array	`val copyOfArray: Array<Int> = intArray.copyOf()`	
Partial copy of Array	`val partialCopyOfArray: Array<Int> = intArray.copyOfRange(0, 2)`	
Lists		
Simple List	`val intList: List<Int> = listOf(1, 2, 3)`	Or `arrayListOf(1,2,3)`
Empty List	`val emptyList: List<Int> = emptyList()`	Or `listOf()`
List with no null elements	`val listWithNonNullElements: List<Int> = listOfNotNull(1, null, 3)`	same as `List(1,3)`
Sets		
Simple Set	`val aSet: Set<Int> = setOf(1)`	Or `hashSetOf(1)` / `linkedSerOf(1)`
Empty Set	`val emptySet: Set<Int> = emptySet()`	Or `setOf()` / `hashSetOf()` / `linkedSetOf()`
Maps		
Simple Map	`val aMap: Map<String, Int> = mapOf("hi" to 1, "hello" to 2)`	Or `mapOf(Pair("hi", 1)` / `hashMapOf("hi" to 1)` / `linkedMapOf("hi" to 1)`
Empty Map	`val emptyMap: Map<String, Int> = emptyMap()`	Or `mapOf()` / `hashMapOf()` / `linkedMapOf()`
Black sheep, mutables		
Simple Mutable List	`val mutableList: MutableList<Int> = mutableListOf(1, 2, 3)`	
Simple Mutable Set	`val mutableSet: MutableSet<Int> = mutableSetOf(1)`	
Simple Mutable Map	`var mutableMap: MutableMap<String, Int> = mutableMapOf("hi" to 1, "hello" 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]`	Return a new iterables with old values + values from added iterable
Minus	`intList - 1`	`[2, 3]`	Returns a new iterables with old values - subtracted one

Operators (cont)

Minus (Iterable)	`intList - listOf(1, 2)`	`[3]	Returns a new iterables with old values - values from subtracted iterable
		Maps	
Plus	`aMap + Pair("Hi", 2)`	`{hi=1, hello=2, Goodbye=3}`	Returns new map with old map values + new Pair. Updates value if it differs
Plus (Map)	`aMap + mapOf(Pair("hello", 2), Pair("Goodbye", 3)`	`{hi=1, hello=2, Goodbye=3}`	Returns new map with old map values + Pairs from added map. Updates values if they differ.
Minus	`aMap - Pair("Hi", 2)`	{Hi=2}	Takes in a key and removes if found
Minus (Map)	aMap - listOf("hello", "hi")	`{}`	Takes in an iterable of keys and removes if found
		Mutables	
Minus Assign	`mutableList -= 2`	`[1, 3]`	Mutates the list, removes element if found. Returns boolean
Plus Assign	`mutableList += 2`	`[1, 3, 2]`	Mutates the list, adds element. Returns boolean
Minus Assign (MutableMap)	`mutableMap.minusAssign("hello")`	{hi=1}	Takes in key and removes if that is found from the mutated map. Returns boolean. Same as `-=`
Plus Assign (MutableMap)	`mutableMap.plusAssign("Goodbye" 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	`intList.associate { Pair(it.toString(), it) }`	`{1=1, 2=2, 3=3}`	Returns a Map containing key-value pairs created by lambda
Map	`intList.map { it + 1 }`	`[2,3,4]`	Returns a new list by transforming all elements from the initial Iterable.
MapNotNull	`intList.mapNotNull { null }`	`[]`	Returned list contains only elements that return as not null from the lamdba

By **Xantier**
cheatography.com/xantier/

Published 14th June, 2017.
Last updated 14th June, 2017.

Transformers (cont)			
MapIndexed	`intList.mapIndexed { 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	`intList.mapIndexedNotNull { idx, value -> if (idx == 0) null else value + 2 }`	`[4,5]`	Combination of Map, MapIndexed & MapIndexedNotNull
MapKeys	`aMap.mapKeys { 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.mapValues { 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	`intList.reversed())`	`[3,2,1]`	
Partition	`intList.partition { it > 2 })`	`Pair([1,2], [3])`	Splits collection into to based on predicate
Slice	`intList.slice(1..2))`	`[2,3]`	Takes a range from collection based on indexes
Sorted	`intList.sorted())`	`[1,2,3]`	
SortedByDescending	`intList.sortedByDescending { it }`	`[3,2,1]`	Sorts descending based on what lambda returns. Lamdba receives the value itself.
SortedWith	`intList.sortedWith(Comparator<Int> { 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.

Transformers (cont)

Flatten	`listOf(intList, aSet).flatten()`	`[2,3,4,1]`	Takes elements of all passed in collections and returns a collection with all those elements
FlatMap with just return	`listOf(intList, aSet).flatMap { 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(intList, aSet).flatMap { iterable: Iterable<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(intList)`	`[(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(intList) { 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	`listOf(Pair("hi", 1), Pair("hello", 2)).unzip()`	`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	`intList.fold(10) { accumulator, value -> accumulator + 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	`intList.foldIndexed(10) { idx, accumulator, value -> if (idx == 2) accumulator else accumulator + value }`	`13` *(10+1+2)*	Accumulates values starting with initial and applying operation from left to right. Lambda receives index as the first value.

Aggregators (cont)			
FoldRight	`intList.foldRight(10) { accumulator, value -> accumulator + 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	`intList.foldRightIndexed(10) { idx, accumulator, value -> if (idx == 2) accumulator else accumulator + value }`	`16` *(10+3+2+1)*	
Reduce	`intList.reduce { accumulator, value -> accumulator + 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	`intList.reduceRight { accumulator, value -> accumulator + 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	`intList.reduceIndexed { idx, accumulator, value -> if (idx == 2) accumulator else accumulator + value }`	`3` *(1+2)*	
ReduceRightIndexed	`intList.reduceRightIndexed { idx, accumulator, value -> if (idx == 2) accumulator else accumulator + value }`	`3` *(2+1)*	
	Grouping		
GroupBy	`intList.groupBy { 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.

Aggregators (cont)			
GroupBy (With new values)	`intList.groupBy({ 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 mutableStringToListMap = mapOf("first" to 1, "second" to 2)` `mutableStringToListMap.values.groupByTo(mutableMapOf<Int, MutableList<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	`intList.groupingBy { it }`
	`.foldTo(mutableMapOf<Int, Int>(), 0) { accumulator, element ->`
	`  accumulator + 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	`intList.groupingBy { "key" }`
	`.aggregate({ key, accumulator: String?, element, isFirst ->`
	`  when (accumulator) {`
	`  null -> "$element"`
	`  else -> accumulator + "$element"`
	`  }`
	`})`
		{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.

Aggregating			
Count	`intList.count()`	3	AKA size
Count (with Lambda)	`intList.count { it == 2 })`	1	Count of elements satisfying the predicate
Average	`intList.average()`	2.0 *((1+2+3)/3 = 2.0)*	Only for numeric Iterables
Max	`intList.max()`	3	Maximum value in the list. Only for Iterables of Comparables.
MaxBy	`intList.maxBy { it * 3 }`	3	Maximum value returned from lambda. Only for Lambdas returning Comparables.

Aggregators (cont)			
MaxWith	`intList.maxWith(oneOrLarger)`	`1`	Maximum value defined by passed in Comparator
Min	`intList.min()`	`1`	Minimum value in the list. Only for Iterables of Comparables.
MinBy	`intList.minBy { it * 3 }`	`1`	Minimum value returned from lambda. Only for Lambdas returning Comparables.
MinWith	`intList.minWith(oneOrLarger)`	`3`	Minimum value defined by passed in Comparator
Sum	`intList.sum()`	`6`	Summation of all values in Iterable. Only numeric Iterables.
SumBy	`intList.sumBy { 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	`intList.sumByDouble { it.toDouble() }`	`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 + simple HOFs			
Method	**Example**	**Result**	**Notes**
	Filtering		
Filter	`intList.filter { it > 2 }`	`[3]`	Filter-in
FilterKeys	`aMap.filterKeys { it != "hello" }`	`{hi=1}`	
FilterValues	`aMap.filterValues { it == 2 }`	`{hello=2}`	
FilterIndexed	`intList.filterIndexed { idx, value -> idx == 2 || value == 2 }`	`[2,3]`	
FilterIsInstance	`intList.filterIsInstance<String>()`	`[]`	All of them are ints
	Taking and Dropping		
Take	`intList.take(2)`	`[1,2]`	Take n elements from Iterable. If passed in number larger than list, full list is returned.
TakeWhile	`intList.takeWhile { it < 3 }`	`[1,2]`	
TakeLast	`intList.takeLast(2)`	`[2,3]`	
TakeLastWhile	`intList.takeLastWhile { it < 3 }`	`[]`	Last element already satisfies this condition --> empty

Filtering and other predicates + simple HOFs (cont)			
Drop	`intList.drop(2)`	`[3]`	Drop n elements from the start of the Iterable.
DropWhile	`intList.dropWhile { it < 3 }`	`[3]`	
DropLast	`intList.dropLast(2)`	`[1]`	
DropLastWhile	`intList.dropLastWhile { it > 2 }`	`[1, 2]`	

Retrieving individual elements			
Component	`intList.component1()`	`1`	There are 5 of these --> `component1()`, `component2()`, `component3()`, `component4()`, `component5()`
ElementAt	`intList.elementAt(2)`	`3`	Retrieve element at his index. Throws IndexOutOfBounds if element index doesn't exist
ElementAtOrElse	`intList.elementAtOrElse(13) { 4 }`	`4`	Retrieve element at his index or return lambda value if element index doesn't exist.
ElementAtOrNull	`intList.elementAtOrNull(666)`	`null`	Retrieve element at his index or return null if element index doesn't exist.
Get (clumsy syntax)	`intList.get(2)`	`3`	Get element by index
Get	`intList[2]`	`3`	Shorthand and preferred way for the one above
GetOrElse	intList.getOrElse(14) { 42 }	`42`	Get element or return lambda value if it doesn't exist.
Get from Map (clumsy syntax)	`aMap.get("hi")`	`1`	
Get from Map	`aMap["hi"]`	`1`	
GetValue	`aMap.getValue("hi")1	`1`	Get value or throw NoSuchElementException
GetOrDefault	`aMap.getOrDefault("HI", 4)`	`4`	Get value or return the value returned from lambda
GetOrPut	`mutableMap.getOrPut("HI") { 5 }`	`5`	MutableMap only. Returns the the value if it exist, otherwise puts it and returns put value.

Finding			
BinarySearch	`intList.binarySearch(2)`	`1`	Does a binary search through the collection and returns the index of the element if found. Otherwise returns negative index.

Filtering and other predicates + simple HOFs (cont)			
Find	`intList.find { it > 1 }`	2	First element satisfying the condition or null if not found
FindLast	`intList.findLast { it > 1 }`	3	Last element satisfying the condition or null if not found
First	`intList.first()`	1	First element of Iterable or throws NoSuchElementException
First with predicate	`intList.first { it > 1 }`	2	Same as find but throws NoSuchElementException if not found
FirstOrNull	`intList.firstOrNull()`	1	Throw safe version of `first()`.
FirstOrNull with predicate	`intList.firstOrNull { it > 1 }`	2	Throw safe version of `first(() -> Boolean)`.
IndexOf	`intList.indexOf(1)`	0	
IndexOfFirst	`intList.indexOfFirst { it > 1 }`	1	
IndexOfLast	`intList.indexOfLast { it > 1 }`	2	
Last	`intList.last()`	3	Throws NoSuchElementException if empty Iterable
Last with predicate	`intList.last { it > 1 }`	3	Throws NoSuchElementException if none found satisfying the condition.
LastIndexOf	`intList.lastIndexOf(2)`	1	
LastOrNull	`intList.lastOrNull()`	3	Throw safe version of `last()`
LastOrNull with predicate	`intList.lastOrNull { it > 1 }`	3	Throw safe version of `last(() -> Boolean)`.

Unions, distincts, intersections etc.

Distinct	`intList.distinct()`	`[1, 2, 3]`	
DistinctBy	`intList.distinctBy { if (it > 1) it else 2 }`	`[1,3]`	
Intersect	`intList.intersect(listOf(1, 2))`	`[1,2]`	
MinusElement	`intList.minusElement(2)`	`[1,3]`	
MinusElement with collection	`intList.minusElement(listOf(1, 2))`	`[3]`	
Single	`listOf("One Element").single()`	`One Element`	Returns only element or throws.
SingleOrNull	`intList.singleOrNull()`	null	Throw safe version of `single()`.
OrEmpty	`intList.orEmpty()`	[1, 2[, 3]	Returns itself or an empty list if itself is null.
Union	`intList.union(listOf(4,5,6))`	`[1,2,3,4,5,6]`	
Union (infix notation)	`intList union listOf(4,5,6)`	`[1,2,3,4,5,6]`	

Cheatography

Kotlin Collection Extensions Cheat Sheet

by Xantier via cheatography.com/38756/cs/12075/

Checks and Actions

Method	Example	Result	Notes
	Acting on list elements		
	`val listOfFunctions = listOf({ print("first ") }, { print("second ") })`		
ForEach	`listOfFunctions.forEach { it() }`	`first second`	
ForEachIndexed	`listOfFunctions.forEachIndexed { idx, fn -> if (idx == 0) fn() else print("Won't do it") }`	`first Won't do it`	
OnEach	`intList.onEach { print(it) }`	`123`	
	Checks		
All	`intList.all { it < 4 }`	`true`	All of them are less than 4
Any	`intList.any()`	`true`	Collection has elements
Any with predicate	`intList.any { it > 4 }`	`false`	None of them are more than 4
Contains	`intList.contains(3)`	`true`	
ContainsAll	`intList.containsAll(listOf(2, 3, 4))`	`false`	
Contains (Map)	`aMap.contains("Hello")`	`false`	Same as `containsKey()`
ContainsKey	`aMap.containsKey("hello")`	`true`	Same as `contains()`
ContainsValue	`aMap.containsValue(2)`	`true`	
None	`intList.none()`	`false`	There are elements on the list
None with predicate	`intList.none { it > 5 }`	`true`	None of them are larger than 5
IsEmpty	`intList.isEmpty()`	`false`	
IsNotEmpty	`intList.isNotEmpty()`	`true`	

<3 Kotlin

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

PDF of this cheat sheet:
http://jussi.hallila.com/Kollections/

Created with <3 by Jussi Hallila

By **Xantier**
cheatography.com/xantier/

Published 14th June, 2017.
Last updated 14th June, 2017.