Kt. Academy

CHEAT SHEET

BASICS

"Hello, World" program

```
fun main(args: Array<String>) {
   println("Hello, World")
}
```

Declaring function

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Single-expression function

```
fun sum(a: Int, b: Int) = a + b
```

Declaring variables

```
val name = "Marcin"  // Can't be changed
var age = 5          // Can be changed
age++
```

Variables with nullable types

```
var name: String? = null
val length: Int
length = name?.length ?: 0
// length, or 0 if name is null
length = name?.length ?: return
// length, or return when name is null
length = name?.length ?: throw Error()
// length, or throw error when name is null
```

CONTROL STRUCTURES

If as an expression

```
fun bigger(a: Int, b: Int) = if (a > b) a else b
```

For loop

```
val list = listOf("A", "B", "C")
for (element in list) {
    println(element)
}
```

When expression

```
fun numberTypeName(x: Number) = when(x) {
    0 -> "Zero"                  // Equality check
    in 1..4 -> "Four or less"    // Range check
    5, 6, 7 -> "Five to seven"   // Multiple values
    is Byte -> "Byte"            // Type check
    else -> "Some number"
}
```

When expression with predicates

```
fun signAsString(x: Int)= when {
    x < 0 -> "Negative"
    x == 0 -> "Zero"
    else -> "Positive"
}
```

CLASSES

Primary constructor

val declares a read-only property, var a mutable one

```
class Person(val name: String, var age: Int)
// name is read-only, age is mutable
```

Inheritance

```
open class Person(val name: String) {
    open fun hello() = "Hello, I am $name"
    // Final by default so we need open
}
class PolishPerson(name: String) : Person(name) {
    override fun hello() = "Dzień dobry, jestem $name"
}
```

Properties with assessors

```
class Person(var name: String, var surname: String) {
    var fullName: String
        get() = "$name $surname"
        set(value) {
            val (first, rest) = value.split(" ", limit = 2)
            name = first
            surname = rest
        }
}
```

Data classes

```
data class Person(val name: String, var age: Int)
val mike = Person("Mike", 23)
```

Modifier `data` adds:

1. `toString` that displays all primary constructor properties

```
print(mike.toString()) // Person(name=Mike, age=23)
```

2. `equals` that compares all primary constructor properties

```
print(mike == Person("Mike", 23)) // True
print(mike == Person("Mike", 21)) // False
```

3. `hashCode` that is based on all primary constructor properties

```
val hash = mike.hashCode()
print(hash == Person("Mike", 23).hashCode()) // True
print(hash == Person("Mike", 21).hashCode()) // False
```

4. `component1`, `component2` etc. that allows deconstruction

```
val (name, age) = mike
print("$name $age") // Mike 23
```

5. `copy` that returns copy of object with concrete properties changed

```
val jake = mike.copy(name = "Jake")
```

Kt. Academy

CHEAT SHEET

COLLECTION LITERALS

```
listOf(1,2,3,4) // List<Int>
mutableListOf(1,2,3,4) // MutableList<Int>

setOf("A", "B", "C") // Set<String>
mutableSetOf("A", "B", "C") // MutableSet<String>

arrayOf('a', 'b', 'c') // Array<Char>

mapOf(1 to "A", 2 to "B") // Map<Int, String>
mutableMapOf(1 to "A", 2 to "B")
// MutableMap<Int, String>

sequenceOf(4,3,2,1) // Sequence<Int>

1 to "A" // Pair<Int, String>

List(4) { it * 2 } // List<Int>
generateSequence(4) { it + 2 } // Sequence<Int>
```

COLLECTION PROCESSING

```
students
   .filter { it.passing && it.averageGrade > 4.0 }
   // Only passing students
   .sortedByDescending { it.averageGrade }
   // Starting from ones with biggest grades
   .take(10) // Take first 10
   .sortedWith(compareBy({ it.surname }, { it.name }))
   // Sort by surname and then name

generateSequence(0) { it + 1 }
// Infinitive sequence of next numbers starting on 0
      .filter { it % 2 == 0 } // Keep only even
      .map { it * 3 } // Triple every one
      .take(100) // Take first 100
      .average() // Count average
```

Most important functions for collection processing

```
val l = listOf(1,2,3,4)
```

filter - returns only elements matched by predicate

```
l.filter { it % 2 == 0 } // [2, 4]
```

map - returns elements after transformation

```
l.map { it * 2 } // [2, 4, 6, 8]
```

flatMap - returns elements yielded from results of trans.

```
l.flatMap { listOf(it, it + 10) } // [1, 11, 2, 12, 3, 13, 4, 14]
```

fold/reduce - accumulates elements

```
l.fold(0.0) { acc, i -> acc + i } // 10.0
l.reduce { acc, i -> acc * i } // 24
```

forEach/onEach - performs an action on every element

```
l.forEach { print(it) } // Prints 1234, returns Unit
l.onEach { print(it) } // Prints 1234, returns [1, 2, 3, 4]
```

partition - splits into pair of lists

```
val (even, odd) = l.partition { it % 2 == 0 }
print(even) // [2, 4]
print(odd) // [1, 3]
```

min/max/minBy/maxBy

```
l.min() // 1, possible because we can compare Int
l.minBy { -it } // 4
l.max() // 4, possible because we can compare Int
l.maxBy { -it } // 1
```

first/firstBy

```
l.first() // 1
l.first { it % 2 == 0 } // 2 (first even number)
```

count - count elements matched by predicate

```
l.count { it % 2 == 0 } // 2
```

sorted/sortedBy - returns sorted collection

```
listOf(2,3,1,4).sorted() // [1, 2, 3, 4]
l.sortedBy { it % 2 } // [2, 4, 1, 3]
```

groupBy - group elements on collection by key

```
l.groupBy { it % 2 } // Map: {1=[1, 3], 0=[2, 4]}
```

distinct/distinctBy - returns only unique elements

```
listOf(1,1,2,2).distinct() // [1, 2]
```

Mutable vs immutable collection processing functions

```
val list = mutableListOf(3,4,2,1)
val sortedResult = list.sorted() // Returns sorted
println(sortedResult) // [1, 2, 3, 4]
println(list) // [3, 4, 2, 1]
val sortResult = list.sort() // Sorts mutable collection
println(sortResult) // kotlin.Unit
println(list) // [1, 2, 3, 4]
```

EXTENSION FUNCTIONS TO ANY OBJECT

Returns Reference to receiver	Receiver	Results of lambda
it	also	let
this	apply	run/with

```
val dialog = Dialog().apply {
   title = "Dialog title"
   onClick { print("Clicked") }
}
```

Kt. Academy

CHEAT SHEET

FUNCTIONS

Function types

`()->Unit` - takes no arguments and returns nothing (`Unit`).

`(Int, Int)->Int` - takes two arguments of type `Int` and returns `Int`.

`(()->Unit)->Int` - takes another function and returns `Int`.

`(Int)->()->Unit` - takes argument of type `Int` and returns function.

Function literals

```
val add: (Int, Int) -> Int = { i, j -> i + j }
// Simple lambda expression

val printAndDouble: (Int) -> Int = {
  println(it)
  // When single parameter, we can reference it using `it`
  it * 2 // In lambda, last expression is returned
}

// Anonymous function alternative
val printAndDoubleFun: (Int) -> Int = fun(i: Int): Int {
  println(i)  // Single argument can't be referenced by `it`
  return i * 2 // Needs return like any function
}

val i = printAndDouble(10) // 10
print(i) // 20
```

Extension functions

```
fun Int.isEven() = this % 2 == 0
print(2.isEven()) // true

fun List<Int>.average() = 1.0 * sum() / size
print(listOf(1, 2, 3, 4).average()) // 2.5
```

DELEGATES

Lazy - calculates value before first usage

```
val i by lazy { print("init "); 10 }
print(i) // Prints: init 10
print(i) // Prints: 10
```

notNull - returns last setted value, or throws error if no value has been set

observable/vetoable - calls function every time value changes. In vetoable function also decides if new value should be set.

```
var name by observable("Unset") { p, old, new ->
   println("${p.name} changed $old -> $new")
}
name = "Marcin"
// Prints: name changed Unset -> Marcin
```

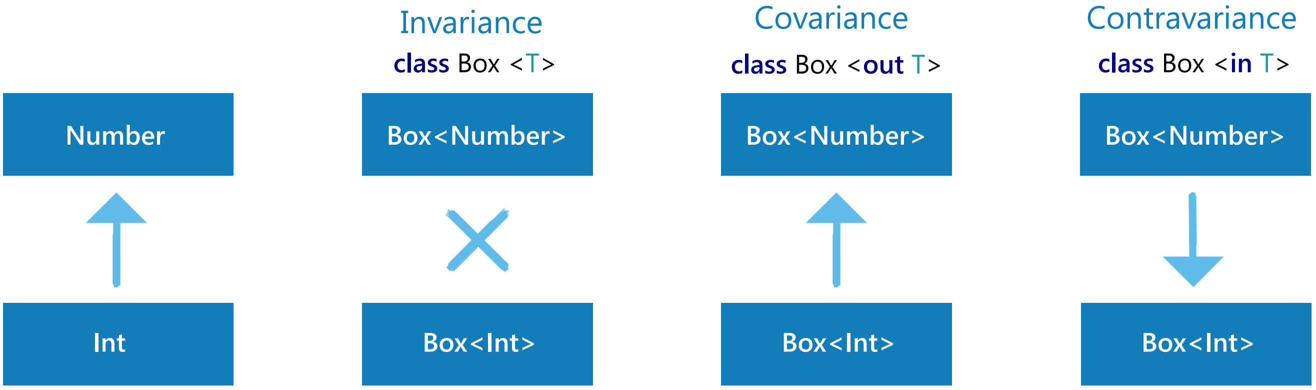
Map/MutableMap - finds value on map by property name

```
val map = mapOf("a" to 10)
val a by map
print(a) // Prints: 10
```

VISIBILITY MODIFIERS

Modifier	Class members	Top-level
Public (default)	Visible everywhere	Visible everywhere
Private	Visible only in the same class	Visible in the same file
Protected	Visible only in the same class and subclasses	Not allowed
Internal	Visible in the same module if class is accessible	Visible in the same module

VARIANCE MODIFIERS



COROUTINES

CoroutineScope

To start coroutine scope you can:
Use `GlobalScope` that has empty coroutine context.
Implement `CoroutineScope` interface.
Create a scope from a context:
with(CoroutineScope(context = context)) **{ ... }**

Coroutine builders

`launch` - Launches new coroutine without blocking current thread and returns a reference to the coroutine as a Job.

`runBlocking` - Runs new coroutine and blocks current thread interruptible until its completion.

`async` - Creates new coroutine and returns its future result as an implementation of Deferred.

`withContext` – Change a coroutine context for some block.

Coroutine context

It is an indexed set of Element instances where every element in this set has a unique Key.

`EmptyCoroutineContext` - Does not change coroutine behavior at all. Like an empty map.

`CoroutineName` - Sets a name of a coroutine for debugging purposes.

`Job` - Lifecycle of a coroutine. Can be used to cancel coroutine. A coroutine is responsible for all children with the same Job. It waits for them and cancels all of them if any had an error (To make children independent use SupervisorJob).

`CoroutineExceptionHandler` - Used to set exception handling for uncaught exceptions.

`ContinuationInterceptor` - Intercepts continuation. Mainly used by dispatchers.

Channels

```
fun CoroutineScope.produceSquares():
    ReceiveChannel<Int> = produce {
       for (x in 1..5) send(x * x)
   }

val squares = produceSquares()
repeat(5) { println(squares.receive()) } // 1, 4, 9, 16, 25

val squares2 = produceSquares()
for(square in squares2) print(square) // 1, 4, 9, 16, 25
```

Coroutine dispachers

`Dispatchers.Default` - Different thread (if possible) It is backed by a shared pool of threads on JVM.

`Dispatchers.Main` - Platform specific main thread (if exists).

`Dispatchers.IO` – Thread designed for offloading blocking IO tasks to a shared pool of threads.

`Dispatchers.Unconfined` - Always uses first available thread (most performant dispatcher).

`newSingleThreadContext` - Creates a new coroutine execution context using a single thread with built-in yield support.

`newFixedThreadPoolContext` - Creates new coroutine execution context with the fixed-size thread-pool and built-in yield support.

Sequence builder

```
val childNumbers = sequence {
   yield(1)
   print("AAA")
   yieldAll(listOf(2, 3))
}
childNumbers.forEach { print(it) } // 1AAA23

val nums = childNumbers.joinToString() // AAA
print(nums) // 1, 2, 3
```

Deal with shared state

`AtomicInteger` - There are atomics for primitives.

`AtomicReference<V>` - Atomic reference.

`Mutex` - Does not let more than one thread at the same time.

```
private val mutex = Mutex()
mutex.withLock { /**/ }
```

Actors

```
sealed class Msg
object IncCounter: Msg()
object PrintCounter: Msg()
class GetCounter(val resp: CompletableDeferred<Int>):Msg()

fun CoroutineScope.counterActor() = actor<Msg> {
   var counter = 0 // Actor state
   for (msg in channel) {
      when (msg) {
         is IncCounter -> counter++
         is PrintCounter -> print(counter)
         is GetCounter -> msg.resp.complete(counter)
      }
   }
}
```