



Kotlin

Table of Contents

1. Declaring Variables
2. Conditional Expressions: If & When
3. Loops
4. Functions
5. OOP
6. Arrays and List
7. Lambda

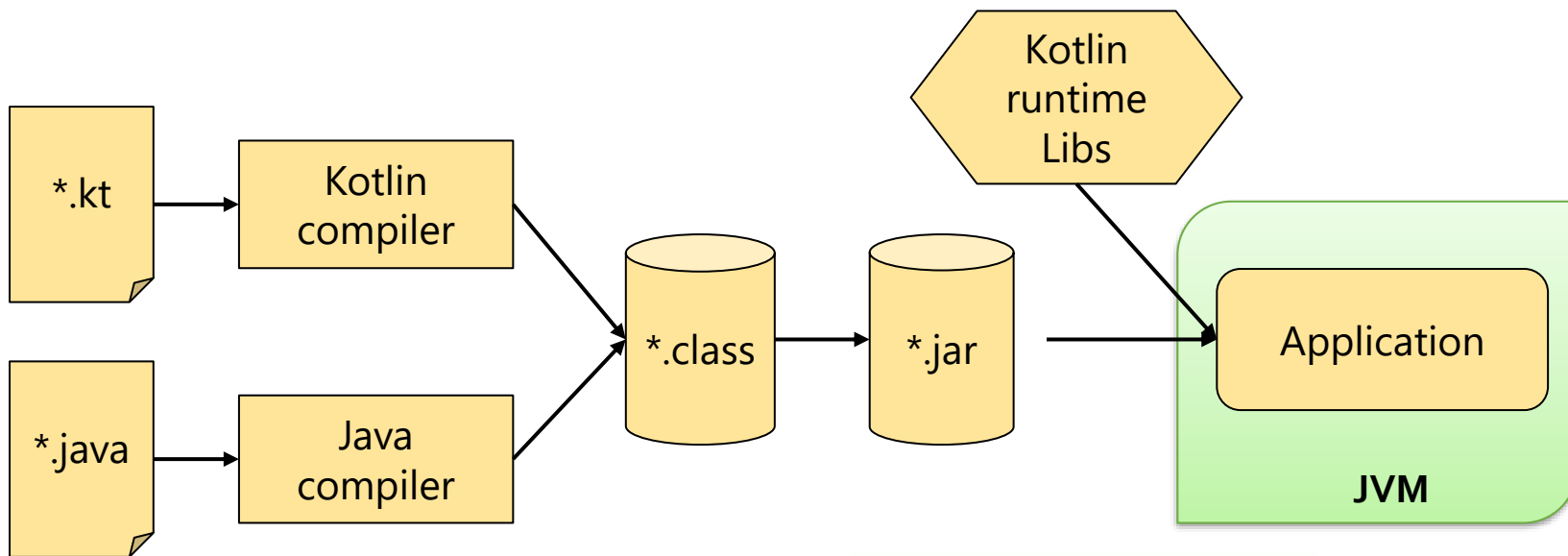
Declaring Variables

Highlights of Kotlin

- Statically typed language: Type **validation** at compile time
- Supports **Type Inference**: type automatically determined from the context
- Much more concise and readable code than Java
- Runs on Java JVM
 - Interoperable with Java code + libraries
 - But can also be compiled to JavaScript and iOS Swift
- Both **functional** and **object-oriented**
- Started in 2011 by JetBrains
 - Kotlin v1.0 was released on 15 February 2016
 - On 7 May 2019, Google announced that the Kotlin as its preferred language for Android app development
 - Current version 1.4 (released August 2020)



How does it work after all?



Can target other platforms



val vs. var

- **val** is **immutable** (read-only) and you can only assign a value to them exactly one time
- **var** is **mutable** and can be reassigned

// val means final - cannot change once initialized

```
val name = "Ali"
```

```
val c: Int
```

```
c = 1
```

//Mutable - can be changed

```
var x = 5
```

```
x += 1
```

/ Type is auto-inferred - The variable datatype is derived from the assigned value */*

```
val city = "Doha"
```

Main Data Types

- Int
- Short
- Long
- Byte
- Double
- Float
- Boolean
- Char
- String
- Any (equivalent to Object in Java)

Strings

//Strings and String Template

```
val firstName = "Ali"  
val lastName = "Faleh"
```

- **String Template** allow creating dynamic templated string with placeholders (instead of string concatenation!)
 - Simple reference uses **\$** and an expression uses **\${}**

```
val fullName = "$firstName $lastName"  
val sum = "2 + 2 = ${2 + 2}"
```

//Multiline Strings

```
val multiLinesStr = """  
    First name: $firstName  
    Last name: $lastName  
    """
```


Convert a number to a string

- Use number's *toString* method

```
val num = 10
```

```
val str = num.toString()
```

Convert a string to a number

- Use string's *toInt* method

```
num = str.toInt()
```

Smart Cast

```
var myVar: Any = "Ali"  
//Smart auto-cast  
if (myVar is String) {  
    println(myVar.last())  
}
```

Nullable Types

- By default variables in Kotlin are **non-nullable**
- Nullable variables are declared **explicitly** to accept a null using **?** after the data type
- **Syntax:**
 - `val iCannotBeNull = "Not Null"`
 - `val iCanBeNull: String? = null`
- `val nullableName: String = null`
 - Compilation Error: Can't assign null to a non-null String
- `val nullableName: String? = null`
 - Compiles ok



Null Safety

- Safe calls:

```
val len = if (name != null) {  
    name.length  
}
```

Safe accessor (?)



```
val len = name?.length
```

- Chaining:

```
student1?.department?.head?.name
```

- **Elvis Operator (?:)**

```
val len = if (name != null) name.length else 0  
// Better syntax is to use the Elvis operator (?:)  
val len = name?.length ?: 0
```

Comments

// slash slash line comment

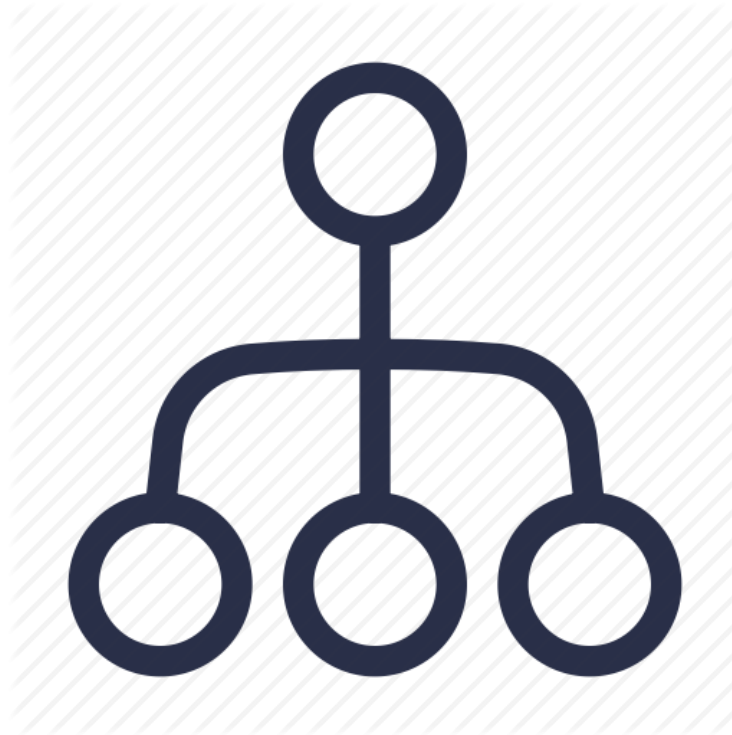
*/**

slash star

block comment

**/*

Control Flow: if, when expressions



if-else expression

```
val age = 20
```

```
// Using 'if' as an expression
```

```
val ageCategory = if (age < 18) {  
    "Teenager"  
} else {  
    "Young Adult"  
}
```

When expression

- Assign a value based on matching condition

```
val month = 8
val season = when (month) {
    12, 1, 2 -> "Winter"
    in 3..5 -> "Spring"
    in 6..8 -> "Summer"
    in 9..11 -> "Autumn"
    else -> "Invalid Month"
}
```

while (...)
do { ... }
for { ... }
Loops

Execute Blocks of Code Multiple Times



While Loop

- While Loop:

```
while (condition) {  
    statements;  
}
```



- Do-While Loop:

```
do {  
    statements;  
}  
while (condition);
```

for Loop Example

```
val names = listOf("Sara", "Fatima", "Ali")
```

```
for (name in names) {  
    println(name)  
}
```

// Loop with index and value

```
for ( (index, value) in names.withIndex()) {  
    println("$index -> $value")  
}
```

Ranges

- Usually defined by: `1..100`
- `1 until 100` // Range excludes 100
- Negative step: `100 downTo 40`
- Any specific step needed?
✓ `100 downTo 40 step 3`

Caution!

```
val notARange = 100 to 40  
// => Pair(100, 40)
```

- To check if a value belongs to the range:

```
val is5inRange = 5 in range2
```

Ranges

```
if (i in 1..10) { // 1 <= i && i <= 10
    println(i)
}

for (i in 1..4) print(i) // "1234"

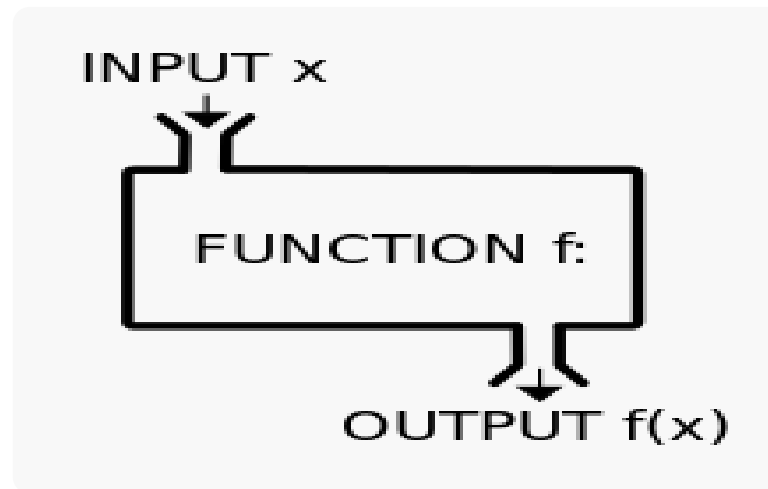
for (i in 1..4 step 2) print(i) // "13"

for (i in 4 downTo 1 step 2)
    print(i) // "42"

for (i in 4..1) print(i) // No Output
for (i in 4 downTo 1)
    print(i) // "4321"

// i in [1, 10), 10 is excluded
for (i in 1 until 10) {
    println(i)
}
```

Functions



Functions

- Can be declared at the **top level** of a file (without belonging to a class)
- Can have a **block or expression body**
- Can have default parameter values to avoid method overloading
- Can use **named** arguments in a function call

```
fun max(a: Int, b: Int): Int { //name - parameters - return type  
|   return if(a>b) a else b //function block body  
}
```

```
fun max(a: Int, b: Int) = if(a>b) a else b //expression body
```

```
max(a = 1, b = 2) //call with named arguments  
max(a: 1, b: 2)
```

Functions

// Function with block body

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

// Function with expression body

// Omit return type

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

//Arrow function - called Lambda expression

```
val sum = { a: Int, b: Int -> a + b }
```

Unit return type

- When defining a function that doesn't return a value, we can use **Unit** as the return type (Unit is equivalent to void in Java)
 - Specifying Unit as a return type is NOT mandatory can omit it

```
fun display(value : Any) : Unit {  
    println(value)  
}
```


Use default parameters instead of overloads

```
fun print() {  
    print(",")  
}
```

```
fun print(separator: String) {  
}
```

```
fun main() {  
    print("|")  
}
```

```
fun print(separator: String = ",") {  
}
```

```
fun main() {  
    print(separator = "|")  
}
```

Extension Method

- Enable adding methods and properties to existing classes

// Extension method extending Int class

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

```
fun main() {  
    val num = 10  
    println("Is $num even: ${num.isEven()}")  
}
```

Infix function calls

- Functions marked with the **infix** keyword can be called using the infix notation (omitting the dot and the parentheses for the call)
- Infix function must satisfy 3 requirements:
 - Must be member function or extension function.
 - Must have a single parameter.
 - The parameter must not accept a variable number of arguments

```
infix fun Int.add(b : Int) : Int = this + b
```

```
fun main() {  
    val x = 10.add(20)  
    val y = 10 add 20    // infix call  
}
```

Exceptions

- Throw:

```
throw Exception("msg")
```

- Handling

```
try {  
}  
catch (e: SomeException) {  
}  
finally {  
}
```

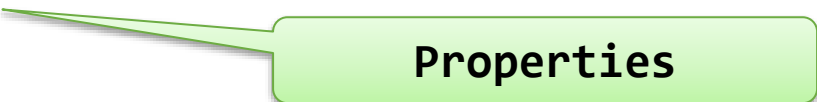
- Expression

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException){ null }
```

OOP

Class

```
class Person(val firstName: String,  
             val lastName: String,  
             val age: Int) {  
    val fullName: String  
        get() = "$firstName $lastName"  
    fun isUnderAge() = age < 18  
}
```



Properties

- **Instantiate:**

```
val student = Person ("Fatima", "Ali", 18)
```

- **Named arguments:**

```
val student = Person (firstName = "Fatima",  
                      lastName = "Ali", 18)
```

Properties are directly accessible without getters / setters

- `val` – read only properties
- `var` – read/write properties
- The primary constructor **cannot** contain any code.

```
class Person(val firstName: String,  
             val lastName: String,  
             var age: Int) {  
    val fullName: String  
        get() = "$firstName $lastName"  
    fun isUnderAge() = age < 18  
}
```

```
val student = Person ("Fatima", "Ali", 18)  
student.age = 20
```

Secondary Constructor

```
class Conference(val name: String,  
                 val city: String,  
                 val isFree: Boolean = false) {  
    var fee : Double = 0.0  
  
    // Secondary Constructor  
    constructor(name: String,  
                 city: String,  
                 fee: Double) : this(name, city) {  
        this.fee = fee  
    }  
}  
  
fun main() {  
    val conference = Conference("Kotlin Conf.", "Doha")  
}
```


Inheritance

```
open class Person( ... ) { ... }
```

```
class Student(firstName: String,  
              lastName: String,  
              age: Int,  
              val gpa: Double  
            ) : Person(firstName, lastName, age) {
```

```
    /*
```

```
    - Override a method from the base class
```

```
    - super keyword to call the implementation of the parent class
```

```
    */  
    override fun toString() = "${super.toString()}. GPA: ${gpa}"
```

```
}
```

- Add **open** keyword to the base class and to properties and methods to be overridden

Data Classes

- Data classes provide autogenerated implementations of **equals()**, **hashCode()**, **copy()** and **toString()** methods

```
data class User(val name: String, val age: Int)
val ali = User(name = "Ali", age = 18)
```

//Copy:

```
val olderAli = ali.copy(age = 19)
val jane = User("Jane", 35)
```

//Destructuring:

```
val (name, age) = jane
// prints "Jane, 35 years of age"
println("$name, $age years of age")
```

Use 'copy' method for data classes

```
class Person(val name: String,  
             var age: Int)  
  
fun happyBirthday(person: Person) {  
    person.age++  
}
```

```
data class Person(val name: String,  
                  val age: Int)  
  
fun happyBirthday(person: Person) =  
    person.copy(  
        age = person.age + 1)
```

No static keyword -> alternatives

- **Top-level functions and properties**
(e.g. for utility classes)
- **Companion objects**
- **object** declaration used to create a **Singleton** (i.e., a single instance for the whole app):
 - Used for declaring the class
 - And providing a single instance of it

```
class Foo {  
    companion object {  
        fun bar() {  
            // ...  
        }  
    }  
}
```

```
object Singleton {  
    fun doSomething() {  
        // ..  
    }  
}
```

Foo.bar()

Singleton.doSomething()

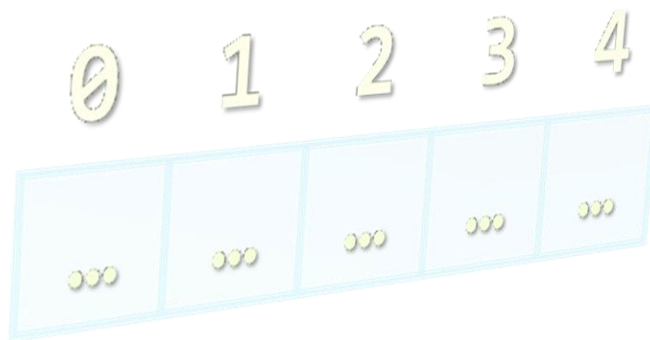
Use `apply` for object initialization

```
fun createLabel(): JLabel {  
    val label = JLabel("Foo")  
    label.foreground = Color.RED  
    label.background = Color.BLUE  
    return label  
}
```

```
fun createLabel() =  
    JLabel("Foo").apply {  
        foreground = Color.RED  
        background = Color.BLUE  
    }
```

Arrays & Lists

Processing Sequences of Elements



Arrays

- Kotlin has a special class called **Array<T>** to declare arrays

```
val names: Array<String> = emptyArray()  
val colors: Array<String> = arrayOf("Red", "Green", "Blue")  
val nulls: Array<String?> = arrayOfNulls(10)
```

```
val numbers: Array<Int> = emptyArray()  
val nums: Array<Int> = arrayOf(2, 3, 4)  
val nullNums: Array<Int?> = arrayOfNulls(10)
```

```
colors.forEach { println(it) }
```

- Better to use **List, Set, Map**

List

// immutable list and mutable list

```
val numsList = listOf(1, 2, 3)
```

```
val mutableNumsList = mutableListof(1, 2, 3)
```

```
mutableNumsList.add(4)
```

```
val sum = numsList.sum() // => 6
```

```
listOf("a", "b", "cc").sumBy { it.length } // => 4
```


List of Objects Example

```
class Car(val brand: String, val age: Int, val horsepower: Int)
```

```
val fleet = listOf(  
    Car(brand: "Ford", age: 1, horsepower: 100),  
    Car(brand: "Mazda", age: 2, horsepower: 120),  
    Car(brand: "Opel", age: 2, horsepower: 95))
```

```
fleet.maxBy { it.horsepower }
```

```
fleet.filter { it.age == 2 }
```

```
fleet.filter { it.age == 2 }.maxBy { it.horsepower }
```

```
fleet.forEach { print("brand: $it.brand") }
```



Chained
Calls

Set

```
// immutable set and mutable set  
  
val colors = setOf("red", "blue", "yellow")  
val mutableColors = mutableSetOf("red", "blue", "yellow")  
mutableColors.add("pink")  
  
val longerThan3 = colors.filter { it.length > 3 }  
  
// => [blue, yellow]
```

Map

```
val languages = mapOf(1 to "Python",  
                      2 to "Kotlin",  
                      3 to "Java")  
  
for ((key, value) in languages) {  
    println("$key => $value")  
}
```

Sequence

// Sequences represent lazily-evaluated collections

```
val numSequence = generateSequence(1, { it + 1 })
```

```
val nums = numSequence.take(10).toList()
```

```
println(nums) // => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

// Convert List to a sequence to enable lazy evaluation

```
val numbers = listOf(1, 2, 3, 4, 5)
```

```
val sum = numbers.asSequence()
```

```
    .map { it * 2 } // Lazy
```

```
    .filter { it % 2 == 0 } // Lazy
```

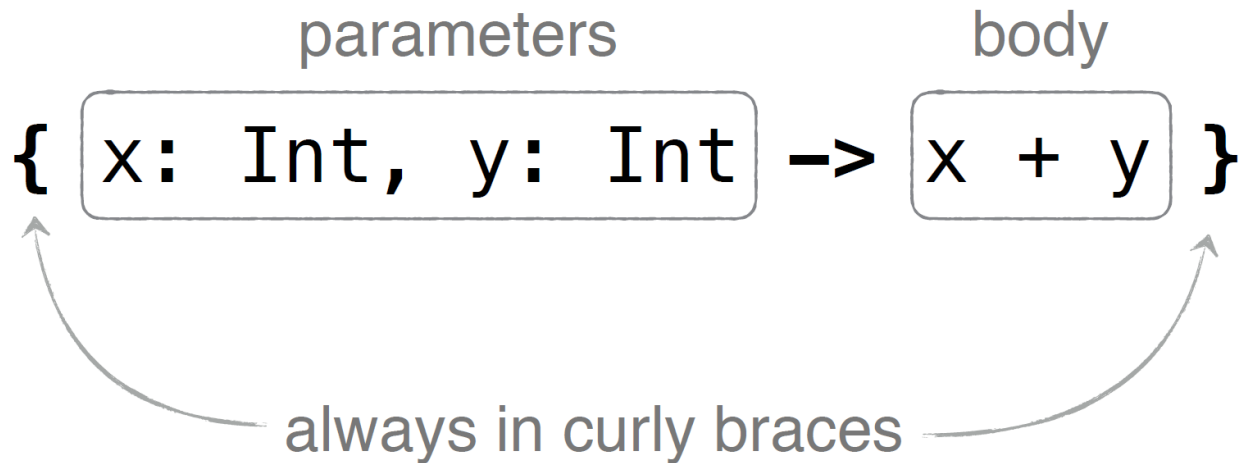
```
    .reduce(Int::plus) // Terminal (eager)
```

```
println(sum) // 30
```

Lambda λ

Lambda

- **Lambda** is a function that you can store them in variables, pass them as parameters, or return them from other functions



`list.any({ i: Int -> i > 0 })`

↑
full syntax

Lambda usage

```
val isEven: (Int) -> Boolean = { i -> i % 2 == 0 }  
val list = listOf(1, 2, 3, 4)  
val evenNumberInList = list.any(isEven)  
val evens = list.filter(isEven)  
  
println("Is there any even number: $evenNumberInList")  
println("Even numbers: $evens")
```

```
list.any() { i: Int -> i > 0 }
```

when lambda is the last argument,
it can be moved out of parentheses

Lambda Shortcut

```
list.any { i: Int -> i > 0 }
```

empty parentheses can be omitted

```
list.any { i -> i > 0 }
```

type can be omitted if it's clear from the context

```
list.any { it > 0 }
```

it denotes an argument (if it's only one)

Multi-line lambda

```
list.any {  
    println("processing $it")  
    it > 0  
}
```

Last expression is the result



Lambda usage

- Allows working with collections in a functional style

e.g. What's an average age of employees working in Doha?

```
val employees = listOf<Employee>(
    Employee("Sara Faleh", "Doha", 30),
    Employee("Mariam Saleh", "Istanbul", 22),
    Employee("Ali Maleh", "Doha", 24)
)

val avgAge = employees.filter { it.city == "Doha" }
    .map { it.age }
    .average()
```

Sort a List using Lambda

```
val names = arrayOf("joe", "ann", "molly", "dolly")  
val sorted = names.sortedBy { name -> name.length }  
  
//Better version  
val sorted = names.sortedBy { it.length }
```

Use 'compareBy' for multi-step comparisons

```
class Person(val name: String,  
             val age: Int)
```

```
fun sortPersons(persons: List<Person>) =  
    persons.sortedWith(  
        compareBy(Person::name,  
                   Person::age))
```

Use 'groupBy' to group items in a collection

```
class Request(val url: String,  
              val remoteIP: String,  
              val timestamp: Long)
```

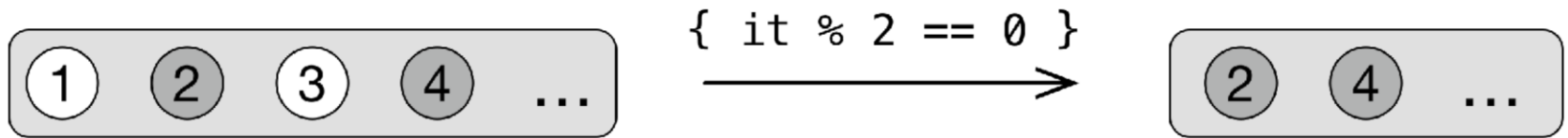
```
fun analyzeLog(log: List<Request>) {  
    val map = log.groupBy(Request::url)  
}
```



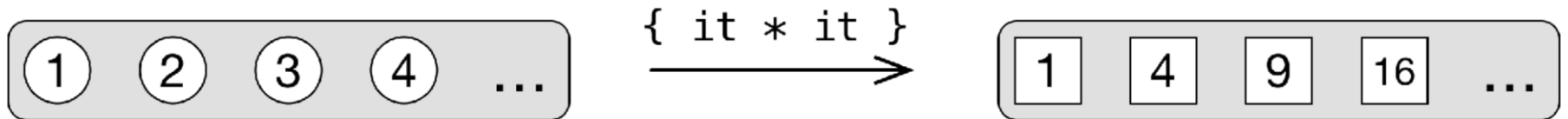
Common operations on collections

- **list.map**
 - Applies a function to each list element
- **list.filter(condition)**
 - Returns a new list with the elements that satisfy the condition
- **list.find(condition)**
- Returns the first list element that satisfy the condition
- **list.reduce**
 - Applies a function against an accumulator and each value of the list to reduce it to a single value.

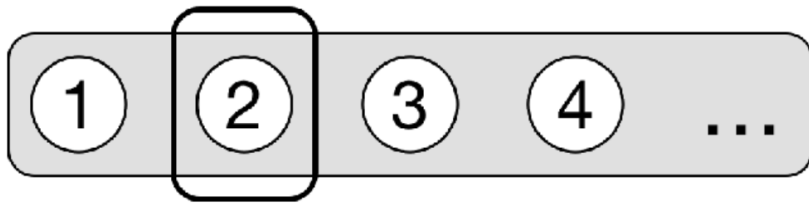
Filter



Map



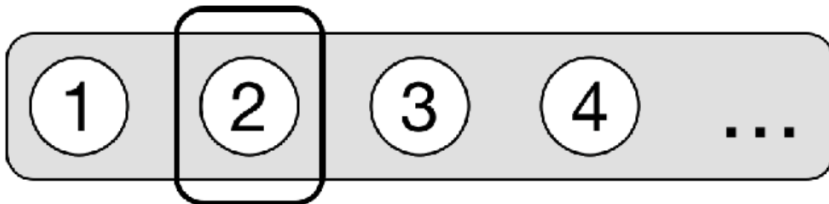
any (all, none)



$\{ it \% 2 == 0 \}$
→

true

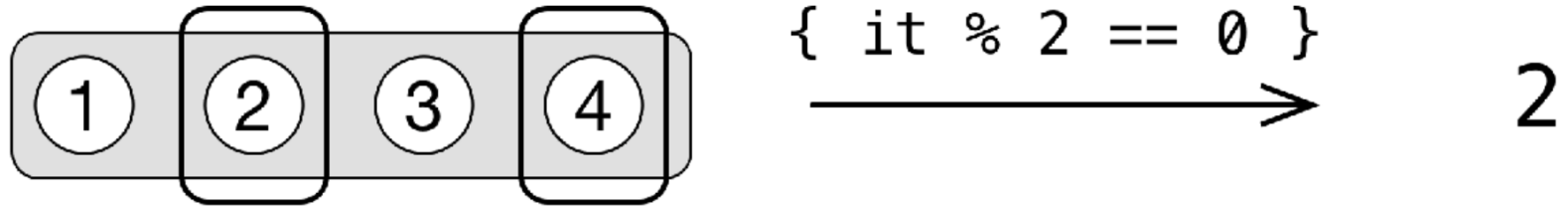
find / firstOrNull



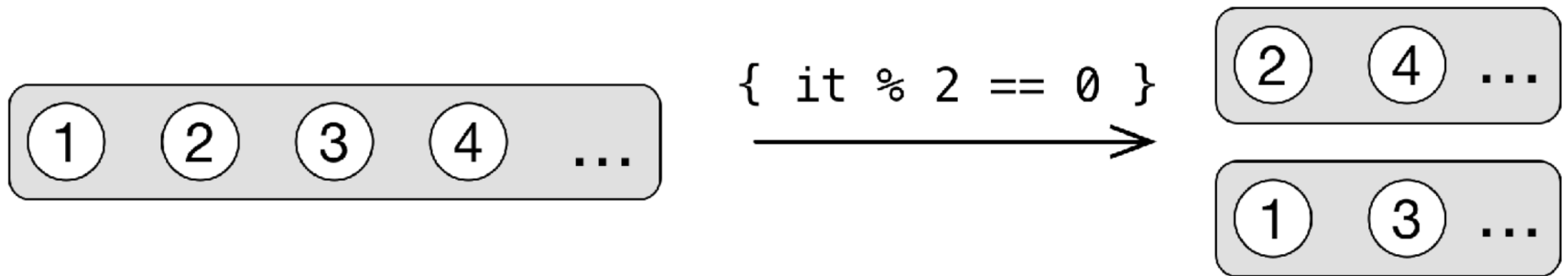
$\{ it \% 2 == 0 \}$
→

2

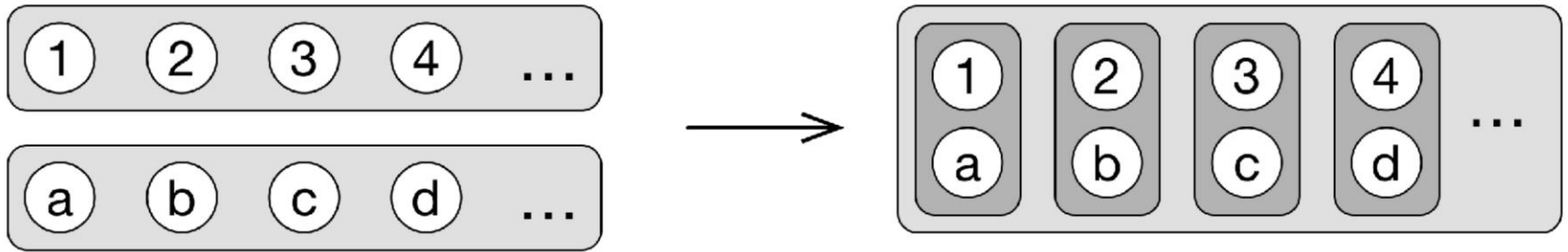
count



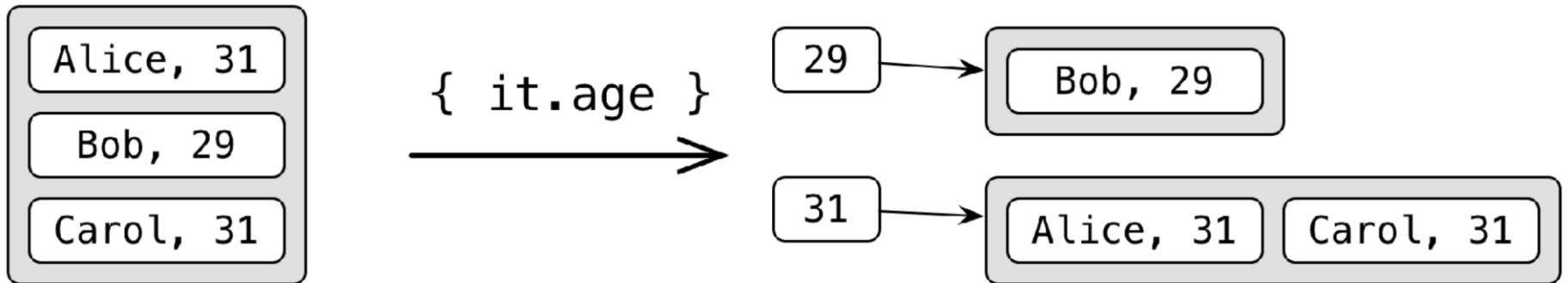
partition



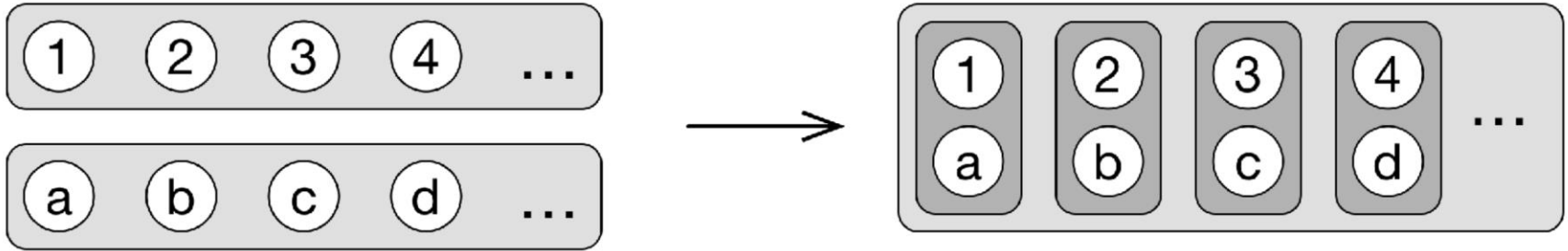
zip



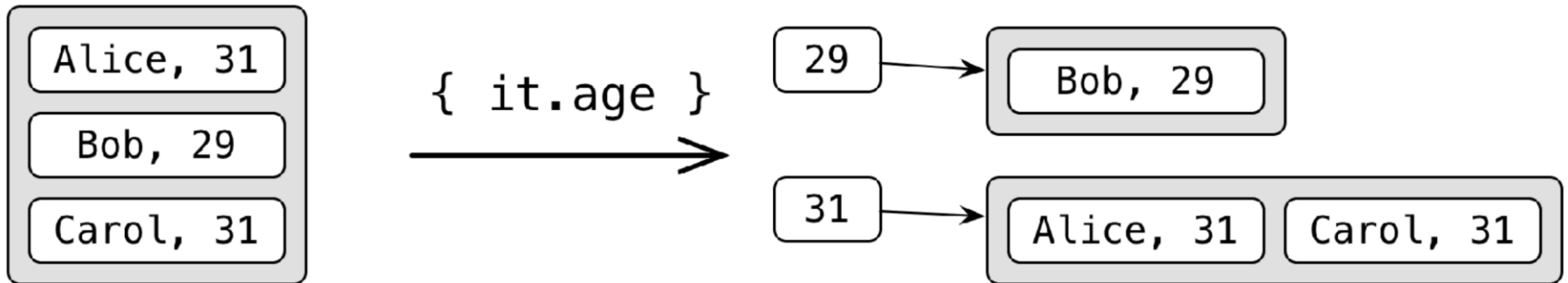
groupBy



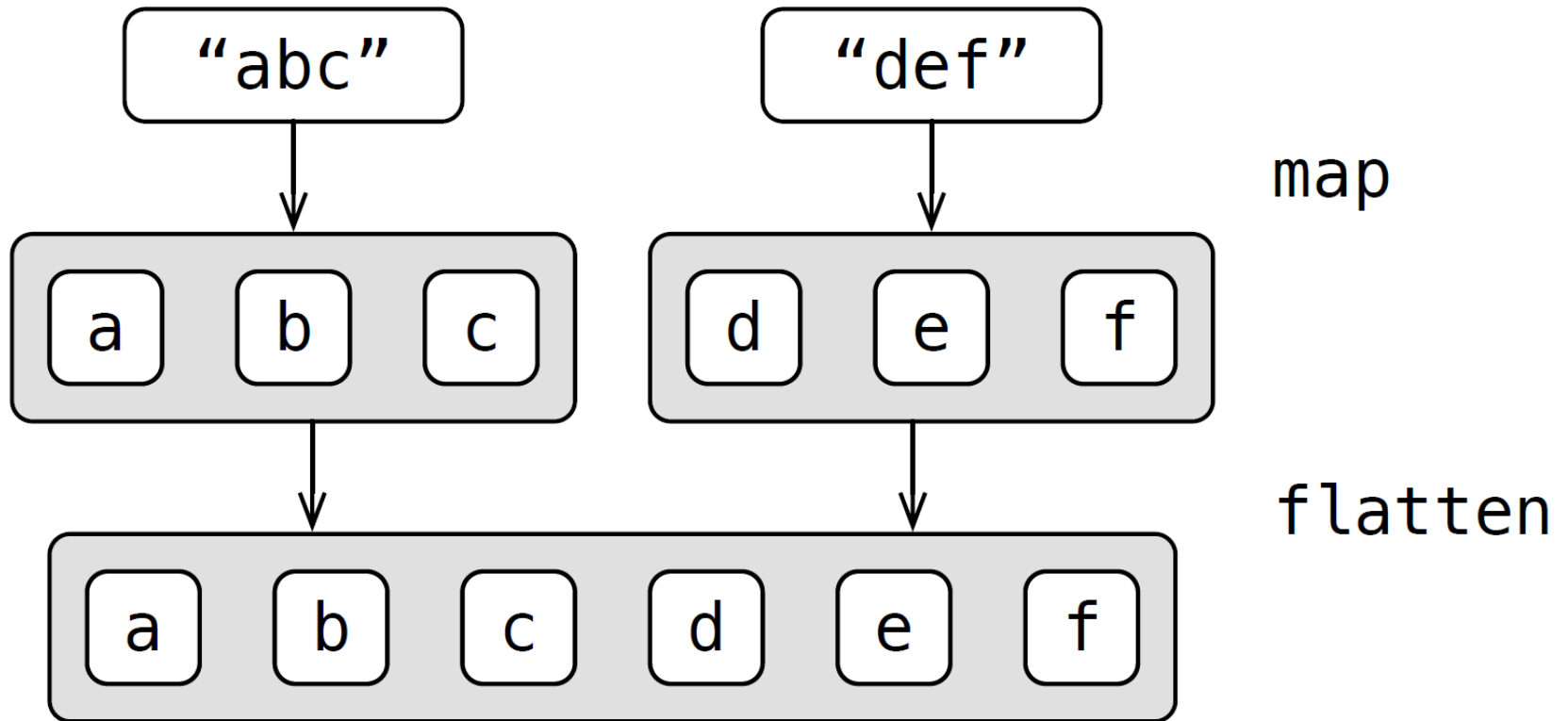
zip



groupBy



flatMap



Kotlin Resources

- Kotlin online courses
 - <https://www.coursera.org/learn/kotlin-for-java-developers>
 - <https://www.udacity.com/course/kotlin-bootcamp-for-programmers--ud9011>
- Kotlin learning resources
 - <https://developer.android.com/courses/kotlin-bootcamp/overview>
 - <https://codelabs.developers.google.com/kotlin-bootcamp/>
 - <https://kotlinlang.org/docs/reference/>