Introduction to Kotlin

https://kotlinlang.org

Part-I Introduction

Kotlin

Kotlin Philosophy

How to use Kotlin

Kotlin

General-purpose programming language

Supports both Object Oriented and Functional programming paradigms

Statically typed and compiled programming language

It is Open Source (Apache 2.0) project

Developed by JetBrains

Kotlin Philosophy

Concise: Drastically reduce the amount of boilerplate code

- Create a POJO with getters, setters, equals(), hashCode(), toString() and copy() in a single line:

```
data class Customer(val name: String, val email: String, val company: String)
```

- Filter a list using a lambda expression:

```
val positiveNumbers = list.filter { it > 0 }
```

Kotlin Philosophy

Safe: Avoid entire classes of errors such as null pointer exception

- Get rid of NullPointerExceptions

```
var output: String
output = null   // Compilation error
```

- Kotlin protects you from mistakenly operating on nullable types

```
val name: String? = null     // Nullable type
println(name.length())       // Compilation error
```

Kotlin Philosophy

Safe: Avoid entire classes of errors such as null pointer exception

- If you check a type is right, the compiler will auto-cast it for you

```
fun calculateTotal(obj: Any) {
    if (obj is Invoice)
        obj.calculateTotal()
}
```

Kotlin Philosophy

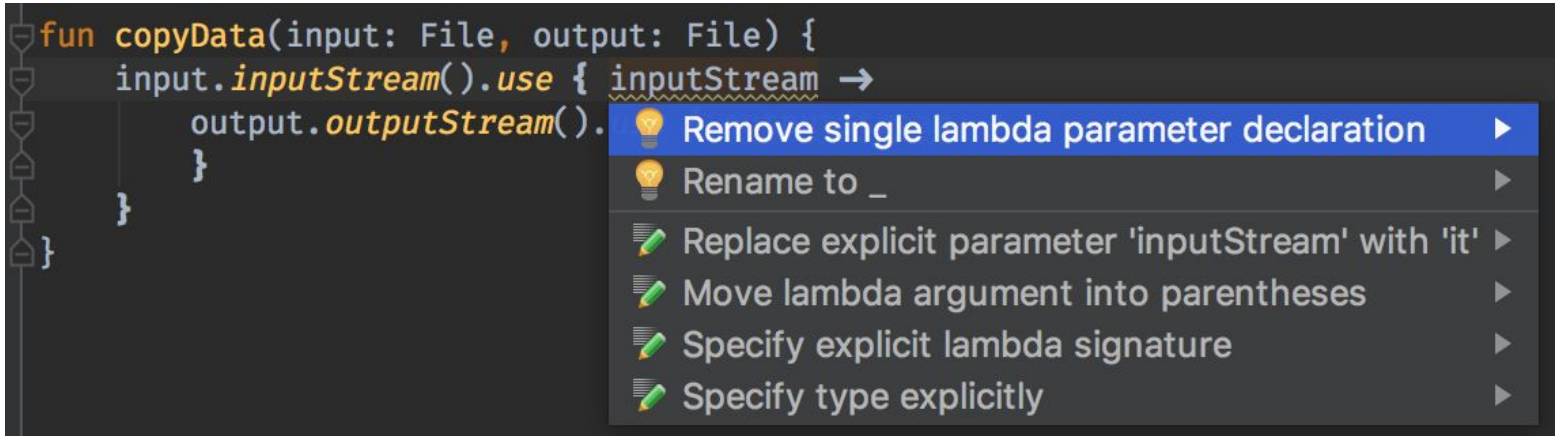
Interoperable: Leverage existing libraries for JVM, Android and the Browser

- Write code in kotline and decide where you want to deploy it

```
import kotlin.browser.window

fun onLoad() {
    window.document.body!!.innerHTML += "<br/>Hello, Kotlin!"
}
```

Kotlin Philosophy

Tool-Friendly: Choose any Java IDE or build from the command line



How to use Kotlin

USE

IntelliJ IDEA

Bundled with Community Edition or IntelliJ IDEA Ultimate

USE

Android Studio

Bundled with Studio 3.0, plugin available for earlier versions

USE

Eclipse

Install the plugin from the Eclipse Marketplace

How to use Kotlin

STANDALONE

Compiler

Use any editor and build from the command line

Downloading the compiler

- Manual Install: Download and unzip the standalone compiler
- SDKMAN!

```
$ curl -s https://get.sdkman.io | bash
$ sdk install kotlin
```

- Snap package

```
$ sudo snap install --classic kotlin
```

https://kotlinlang.org/docs/tutorials/command-line.html

How to use Kotlin

STANDALONE

Compiler

Use any editor and build from the command line

Compiling and Running Kotlin application (CMD)

1. Write your Kotlin code in a file (**.kt**) (e.g. hello.kt)
2. Compile the application using the Kotlin compiler

```
$ kotlinc hello.kt -include-runtime -d hello.jar
```

3. Run the application

```
$ java -jar hello.jar
```

https://kotlinlang.org/docs/tutorials/command-line.html

How to use Kotlin

STANDALONE

Compiler

Use any editor and build from the command line

Running the REPL (Read-Eval-Print Loop)

We can run the compiler without parameters to have an interactive shell

```
[Ocean] ~/tutorials/kotlin/command_line/kotlinc$ bin/kotlinc-jvm
Kotlin interactive shell
Type :help for help, :quit for quit
>>> 2+2
4
>>> println("Welcome to the Kotlin Shell")
Welcome to the Kotlin Shell
>>>
```

https://kotlinlang.org/docs/tutorials/command-line.html

How to use Kotlin

STANDALONE

Compiler

Use any editor and build from the command line

Using the command line to run scripts

Kotlin can also be used as a scripting language.

A script is a Kotlin source file (**.kts**) with top level executable code

```
import java.io.File

val folders = File(args[0]).listFiles { file -> file.isDirectory() }
folders?.forEach { folder -> println(folder) }
```

To run the script

```
$ kotlinc -script list_folders.kts <path_to_folder_to_inspect>
```

https://kotlinlang.org/docs/tutorials/command-line.html

PART-II Basic

Hello World in Kotlin

Packages and Functions

Variables and Comments

Control structures

Exceptions

Extension functions

Hello World in Kotlin

```
package intro
fun main(args: Array<String>) {
    val name = if (args.size > 0) args[0] else "World"
    println("Hello, $name!")
}
```

```
package intro
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Hello World in Kotlin

Observation

- No class definition
  - You can define functions at the top level
- if is an expression not a statement
- Array is class
- String template
  - Easy way of accessing variable values inside string literals
- No semicolon is required

```
package intro

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Define Packages

```
package et.edu.aait.myproject

import java.util.*
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

Functions

Function having two Int parameters with Int return type:

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

Calling **`sum`** function from **`main`** function

```
fun main() {
    println(sum(2, 3))
}
```

Functions

Function with an expression body and **`inferred return type`**

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

Use **`Unit`** as return type of functions that return no meaningful value

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

`Unit` is optional, it can be removed

Functions

Top-Level Functions

```
package et.edu.aait.introproject

fun areaOfSquare(size: Int): Int{
    return size*size
}
```

Member Functions

```
package et.edu.aait.introproject

class Square(val size:Int){
    fun area() = size * size
}
```

Local Functions

```
package et.edu.aait.introproject

fun outerFunction(){

    fun localFunction(){
        println("This is local function")
    }
    localFunction()
    println("This is outer function")
}
```

Functions

Named arguments

```
package et.edu.aait.introproject

fun addPrefixPostfix(pre:String, post:String, word:String):String{
    return pre+word+post
}

fun main(){
    println(addPrefixPostfix("left-","-right","middle"))
    println(addPrefixPostfix(post="-right", word="middle",pre="left-"))
}
```

Functions

Default Argument

```
fun addPrefixPostfix(
    pre: String ="-right",
    post:String = "left-",
    word:String):String{
    return pre+word+post
}

fun main(){
    println(addPrefixPostfix(word="middle"))
    println(addPrefixPostfix(post="-low", word="middle",pre="high-"))
}
```

Functions

Variable Arguments

```
fun printList(vararg list:String) {
    list.forEach { item -> println(item) }
}

fun main() {
    printList("a", "b", "c","4")
}
```

Function Types

A function type definition consists of two parts:

- the function's parameters, in parentheses, followed by
- its return type, delimited by the arrow (->)

`() -> String` tells the compiler what kind of function a variable can hold

```
fun main(){

    val greetingFunction: () -> String = {
        val currentYear = 2019
        "Welcome to Kotlin $currentYear"
    }

    print(greetingFunction())
}
```

Variables

Use keyword **val** to define **read-only (immutable)** variables. They can be assigned a value only once

```
fun main() {

    val a: Int = 1  // immediate assignment
    val b = 2   // `Int` type is inferred
    val c: Int  // Type required when no initializer is provided
    c = 3       // deferred assignment

    println ("The sum of $a, $b, $c is ${a + b + c}")
}
```

Variables

Use keyword **var** to define **mutable** variables. They can be re-assigned a value

```
fun main() {

    var x = 5 // `Int` type is inferred
    println ("Initial value of x is $x")
    x += 1
    println ("The new value of x is $x")

}
```

Variables

Top-level variables

```
package et.edu.aait.introproject

val PI = 3.14

fun areaOfCircle(radius: Int) = PI * radius * radius

fun main() {

    println("The area of circle with radius 3 is ${areaOfCircle(3)}")

}
```

Comments

Just like Java and JavaScript, Kotlin supports end-of-line and block comments.

Unlike Java, block comments in Kotlin can be nested.

```
// This is an end-of-line comment

/* This is a block comment
   on multiple lines.

    /*
       This is nested comment
     */

 */
```

Control structures

Conditionals: if expression

Note that if is an expression not a statement

```
fun main(args: Array<String>) {

    val arg1 = Integer.parseInt(args[0])
    val arg2 = Integer.parseInt(args[1])
    val max:Int = if (arg1 > arg2) arg1 else arg2
    println ("The maximum of $arg1 and $arg2 is $max")

}
```

Control structures

Conditionals: when

```
fun checkValue(value: Int){
    when (value) {
        1 -> print("value == 1")
        2 -> print("value == 2")
        else -> { // Note the block
            print("value is neither 1 nor 2")
        }
    }
}
fun main(){
 checkValue(2)
}
```

Control structures

Conditionals: when

```
fun checkValue(value: Any){
    when (value) {
        0,1,2 -> print("value is 0 or 1 or 2")
        is String -> print("The size of the string is ${value.length}")
        else -> {
            print("value is not 0, 1, 2 or a String")
        }
    }
}
```

Control structures

Conditionals: when

When is also expression

```
fun convertGrade(grade: Char) = when(grade){
    'A' -> 4
    'B' -> 3
    'C' -> 2
    'D' -> 1
    'F' -> 0
    else -> -1
}

fun main() {
    println(convertGrade('C'))
}
```

Control structures

Conditionals: when

When without argument

```
fun compareValues(a: Int, b: Int) = when {
        a > b -> println("a is greater than b")
        else -> println("a is equal to b or less than b")
}

fun main() {
    compareValues(4,3)
}
```

Control structures

Loops: for

```
fun main() {
    val list = listOf(1,2,3,4)
    for (l in list ){
        println( "$l times 2 is ${l*2}")
    }
}
```

Control structures

Loops: for

```
fun main() {
    val map:Map<Int, String> = mapOf(2 to "two", 3 to "three")
    for ((key, value) in map ){
        println( "key=$key maps to value=$value ")
    }
}
```

Control structures

Loops: for

```
fun main() {
    val list = listOf("Nine", 1, 2, 3, "three")
    for ((index, element) in list.withIndex()){
        println("Index=$index, element=$element")
    }
}
```

Control structures

Loops: for

Upper bound included

```
fun main() {
    for (i in 1..9){
        println(i)
    }
}
```

Upper bound not included

```
fun main() {
    for (i in 1 until 10){
        println(i)
    }
}
```

Control structures

Loops: for

Reverse count

```
fun main() {
    for (i in 10 downTo 1){
        println(i)
    }
}
```

Reverse count with step value

```
fun main() {
    for (i in 10 downTo 1 step 2){
        println(i)
    }
}
```

Control structures

Loops: for

Iterating over String

```
fun main() {
    for ( c in "hello"){
        print(c)
    }
}
```

Control structures

Note the use of in

it can be used for iteration and also to check belongingness

Iterating over String

```
fun main() {
    for ( c in "hello"){
        print(c)
    }
}
```

Checking for belonging

```
fun isSmallLetter(c:Char) =  c in 'a'..'z'
```

Exceptions

throw is an expression

```
fun checkMark(value: Int) = when (value) {
    in 90..100 -> "Excellent"
    in 70..89 -> "Very Good"
    in 50..69 -> "Good"
    in 0..49 -> "fail"
    else -> throw IllegalArgumentException("Invalid input: $value")
}
```

Exceptions

try is an expression

```
fun main() {
    stringToInt("four")
}

fun stringToInt(value:String){
   val result =  try {
      Integer.parseInt(value)
    } catch (e: NumberFormatException){e}
    println(result)
}
```

Extension Functions

```
fun String.lastChar() = this[this.length-1]
fun String.firstChar() = this[0]

fun main() {
    println("abcd".lastChar())
    println("abcd".firstChar())
}
```

PART-III Classes and Objects

Classes and Constructors

Modifiers

Interfaces

Inheritance

Properties

Special Classes

Object, Object Expression and Companion Object

Classes and Constructors

```
//  A class without any properties or user-defined constructors
class Customer

/* a class with two properties: immutable id and mutable email,
and a constructor with two parameters */
class Contact(val id: Int, var email: String)
```

Classes and Constructors

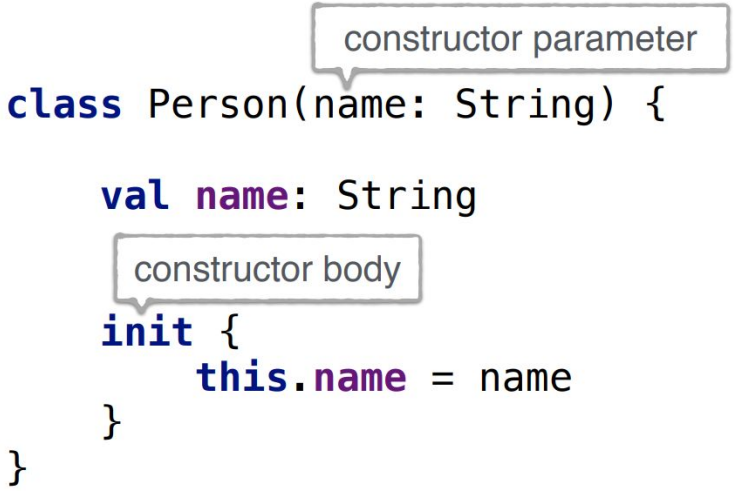
```
fun main() {

    // Creates an instance of the class Customer via the default constructor
    val customer = Customer()

    /* Creates an instance of the class Contact
    using the constructor with two arguments */
    val contact = Contact(1, "mary@gmail.com")

    // Accesses the property id
    println(contact.id)
    // Updates the value of the property email
    contact.email = "jane@gmail.com"
}
```

Classes and Constructors

Concise Primary Constructor

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

Full Constructor



Classes and Constructors

val/var on a parameter creates a property

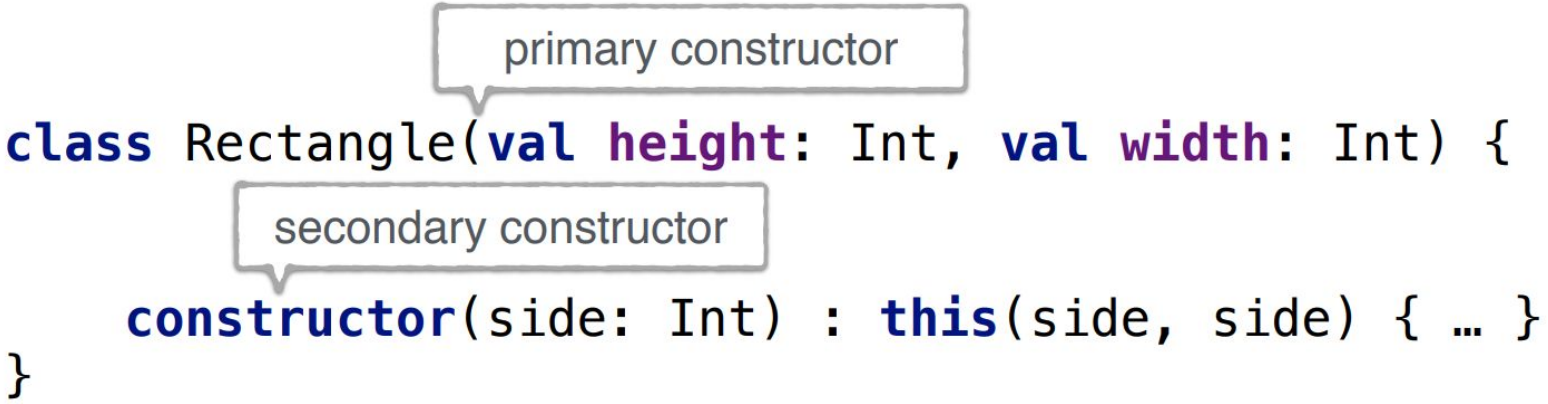
```
class Person(name: String) {

    val name: String

    init {
        this.name = name
    }
}
```

=

```
class Person(val name: String)
```

Classes and Constructors

Secondary Constructor



this(...) calls another constructor of the same class

Modifiers

final, open, abstract, override

public, private, internal, protected

Definition:

A module: *a set of Kotlin files compiled together*

Modifiers

final (used by default): cannot be overridden

open: can be overridden

abstract: must be overridden (can’t have an implementation)

override (mandatory): overrides a member in a superclass or interface

Modifiers

Modifier	Class Member	Top-level Declaration
public	Visible everywhere	Visible everywhere
internal	Visible in a module	Visible in a module
protected	Visible in a subclass	----------
private	Visible in a class	Visible in a file

Change Visibility

```
class InternalComponent
internal constructor(name: String) {
    …
}
```

Example

```
class Rectangle(val width:Int, val height:Int) {
    constructor(side: Int):this(side, side)
}

fun main() {
    val rectangle = Rectangle (2, 3)
    val square = Rectangle(2)
    println(rectangle.width)
    println(rectangle.height)
    println(square.height)
    println(square.width)
}
```

Interfaces and their Implementation

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

Interfaces Example

```
interface MotorVehicle {
    fun moveUp(distance: Double)
    fun moveDown(distance: Double)
}

class Car(var position: Double): MotorVehicle {
    override fun moveUp(distance: Double){
        position += distance
    }
    override fun moveDown(distance: Double){
        position -= distance
    }
}
```

Inheritance

The same syntax can be used for extending a class & implementing an interface

```
interface Base
class BaseImpl : Base

open class Parent
class Child : Parent()
```

constructor call

Inheritance

Calling a constructor of the parent class

```
open class Parent(val name: String)
class Child(name: String) : Parent(name)

open class Parent(val name: String)
class Child : Parent {
    constructor(name: String, param: Int) : super(name)
}
```

Inheritance Example

```
open class Shape{
    open fun area(): Double = 0.0
}

class Square (private val size: Double): Shape() {
    override fun area() =  size*size
}
```

```
fun main() {
    val square = Square(2.2)
    println("Area of Square with size 2.2 is ${square.area()}")
}
```

Properties

Properties = fields + accessors

Read Only Properties = fields + getters

Mutable Properties = fields + getters + setters

```
class Contact(val name:String, val address:String)

fun main() {
    val contact1 = Contact("email", "betsegawlemma@gmail.com")
    val contact2 = Contact("office", "Samsung blg 139")
    println("Address ${contact1.name} ${contact1.address}")
    println("Address ${contact2.name} ${contact2.address}")
}
```

Properties

Properties = fields + accessors

Read Only Properties = fields + getters

Mutable Properties = fields + getters + setters

```
class Contact(val name:String, var address:String)

fun main() {
    val contact1 = Contact("email", "betsegawlemma@gmail.com")
    contact1.address = "betsegaw.lemma@aait.edu.et"
    println("Address ${contact1.name} ${contact1.address}")
}
```

Properties

Custom getters

```
class Rectangle(private val height:Int, private val width:Int){

    val isSquare:Boolean
        get(){
            return height==width
        }
    val area:Int
        get(){
            return height*width
        }
}

fun main() {
  val rect1 = Rectangle(3,3)
    println("Is the rectangle square? ${rect1.isSquare}")
    println("The area of the rectangle is: ${rect1.area}")
}
```

Properties

Private setters

```
class Rectangle(private val height:Int, private val width:Int){

    var isSquare:Boolean = false
        get(){
            return height==width
        }
        private set
    var area:Int = 0
        get(){
            return height*width
        }
        private set
}

fun main() {
  val rect1 = Rectangle(3,3)
    println("Is the rectangle square? ${rect1.isSquare}")
    println("The area of the rectangle is: ${rect1.area}")
}
```

Properties

Custom setters

```
class Switch{

    var state: String = "OFF"
        set(value){
            println("State changed ${field->$value")
            field = value
        }
}

fun main() {
  val switch = Switch()
    switch.state = "ON"
}
```

Data Classes

Used for holding data

```
data class User(val name: String, val age: Int)
```

The compiler automatically derives members such as the following from all properties declared in the primary constructor

- equals()/hashCode()
- toString() of the form "User(name=Abebe, age=42)"
- copy() function

Note that the compiler only uses the properties defined inside the primary constructor for the automatically generated functions

Data Classes

have to fulfill the following requirements:

```
data class User(val name: String, val age: Int)
```

- The primary constructor needs to have at least one parameter
- All primary constructor parameters need to be marked as val or var
- Data classes cannot be abstract, open, sealed or inner

Data Classes: Copying

```
data class Student(val name: String, val id: String, val department: String)

fun main() {
    val abebe = Student("Abebe", "ATR/0000/00", "ITSC")
    val aster = abebe.copy("Aster", "ATR/1111/11")
    println("Abebe's Info: $abebe")
    println("Aster's Info: $aster")
}
```

Enum Classes

```
enum class Color{
    BLUE, ORANGE, RED
}

fun getDescription(color: Color) = when(color) {
    Color.BLUE -> "Cold"
    Color.ORANGE -> "Mild"
    Color.RED -> "Hot"
}
```

Enum Classes

```
enum class Color{
    BLUE, ORANGE, RED
}

fun getDescription(color: Color) = when(color) {
    Color.BLUE -> "Cold"
    Color.ORANGE -> "Mild"
    Color.RED -> "Hot"
}
```

Enum Classes

```
enum class Color{
    BLUE, ORANGE, RED
}

fun getDescription(color: Color) = when(color) {
    Color.BLUE -> "Cold"
    Color.ORANGE -> "Mild"
    Color.RED -> "Hot"
}
```

Enum Classes (with properties and methods)

```
enum class Color(val r: Int, val g: Int, val b: Int){
    BLUE(0,0,255), ORANGE(255,165,0), RED(255,0,0);
    fun rgb() = (r * 256 + g) * 256 + b
}

fun main(){
    println(Color.ORANGE.g)
    println(Color.BLUE.rgb())
}
```

Enum Classes (with properties and methods)

```
import Color.*
enum class Color(private val r: Int, val g: Int, val b: Int){
    BLUE(0,0,255), ORANGE(255,165,0), RED(255,0,0);
    fun rgb() = (r * 256 + g) * 256 + b
}

fun main(){
    println(ORANGE.g)
    println(BLUE.rgb())
}
```

Sealed Classes

Are used for representing restricted class hierarchies in which an object can only be of one of the given types

Are, in a sense, an extension of enum classes

can have subclasses, but all of them must be declared in the same file

A subclass of a sealed class can have multiple instances which can contain state

Sealed Classes

You can declare the subclasses inside the sealed class or outside but they always have to be declared in the same file.

Is abstract by itself, it cannot be instantiated directly and can have abstract members

Are not allowed to have non-private constructors (their constructors are private by default)

Sealed Classes: Example

```
sealed class Operation

data class Add(val leftOp: Double, val rightOp: Double): Operation()
data class Subtract(val leftOp: Double, val rightOp: Double): Operation()
data class Multiply(val leftOp: Double, val rightOp: Double): Operation()
data class Divide(val leftOp: Double, val rightOp: Double): Operation()

fun evaluate(operation: Operation) = when(operation){
    is Add -> operation.leftOp + operation.rightOp
    is Subtract -> operation.leftOp - operation.rightOp
    is Multiply -> operation.leftOp * operation.rightOp
    is Divide -> operation.leftOp / operation.rightOp
}
```

Sealed Classes: Example

```
sealed class Expr
data class Num(val number: Double): Expr()
data class Add(val op1: Expr, val op2: Expr): Expr()

fun evaluate(expr: Expr): Double = when (expr) {
    is Num -> expr.number
    is Add -> evaluate(expr.op1) + evaluate(expr.op2)
}

fun main() {
    println(evaluate(Add(Num(2.0), Num(4.0))))
}
```

Object Declaration

With the object keyword, you specify that a class will be limited to a single instance – a singleton

A single Counter instance (object)

```
object Counter{
    var count:Int = 0
    private set
    fun updateCount() = ++count
}
fun main(){
    println(Counter.updateCount())
    println(Counter.updateCount())
}
```

Object Declaration

With the object keyword, you specify that a class will be limited to a single instance – a singleton

A single Game instance (object)

```
fun main() {
    Game.play()
}

object Game {
    init {
        println("Welcome, adventurer.")
    }
    fun play() {
        while (true) {
            // Play Game
        }
    }
}
```

Object Expressions

Replaces Java anonymous class

```
window.addMouseListener(
        object : MouseAdapter() {
            override fun mouseClicked(e: MouseEvent) {
                // ...
            }

            override fun mouseEntered(e: MouseEvent) {
                // ...
            }
        }
)
```

Object Expressions

Replaces Java anonymous class

```
interface Vehicle {
    fun drive(): String
}

fun start(vehicle: Vehicle) = println(vehicle.drive())

fun main() {
    start(object : Vehicle {
        override fun drive() = "Driving really fast"
    })
}
```

Companion Object

It is an object that is common to all instances of that class.

It is similar to static fields in Java.

```
class Car(val color: String,val model: String) {
    companion object Factory {
        val cars = mutableListOf<Car>()

        fun makeCar(color: String, model: String): Car {
            val car = Car(color,model)
            cars.add(car)
            return car
        }
    }
}

fun main() {
    Car.makeCar("Blue", "BMW 2-Series")
    Car.makeCar("Red", "Chevrolet Bolt")
    Car.cars.forEach{ car -> println("Model: ${car.model}, Color = ${car.color}") }
}
```

Part-IV

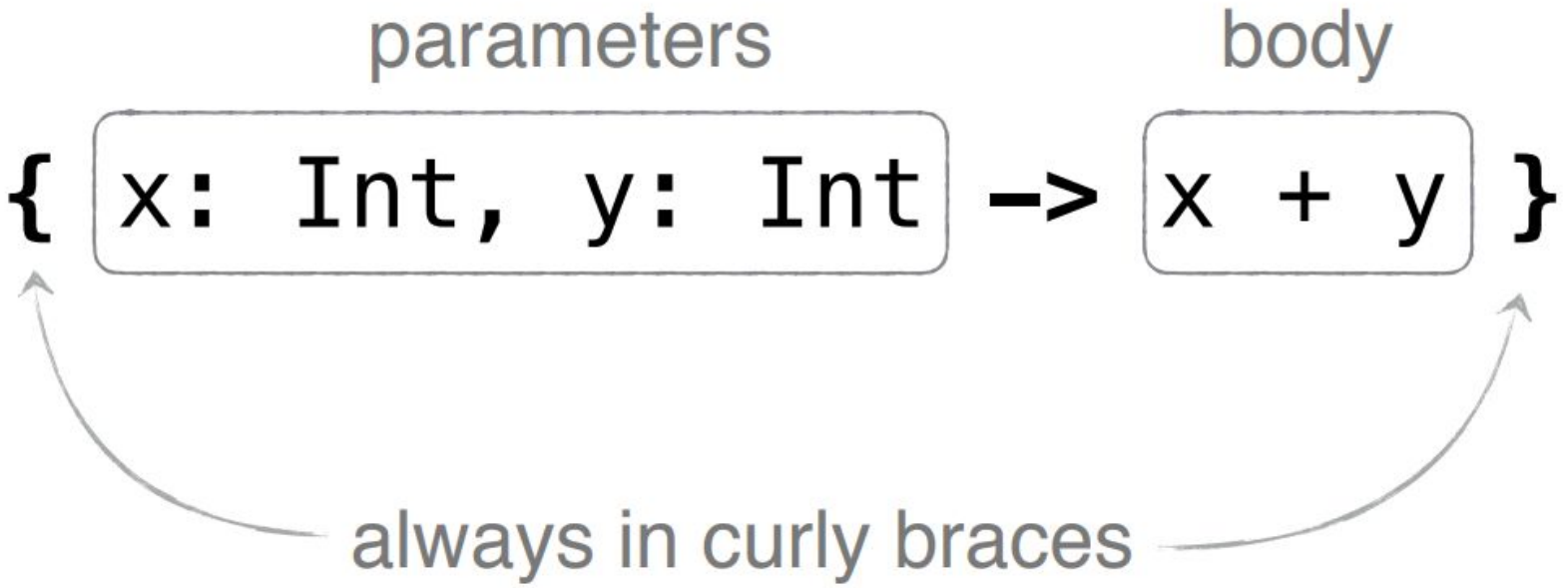
Lambda Expression

Operations on Collections

Nullability

Safe casts

Lambda Expression: Syntax



Lambda Expression: Example

Full Syntax

```
fun anyEven1() {
    val list = listOf(2, 3, 4, 6)
    val result = list.any({ item: Int -> item % 2 == 0 })
    println(result)
}
```

Lambda Expression: Example

When lambda is the last argument, it can be moved out of parenthesis

```
fun anyEven2() {
    val list = listOf(2, 3, 4, 6)
    val result = list.any() { item: Int -> item % 2 == 0 }
    println(result)
}
```

Lambda Expression: Example

Empty parenthesis can be omitted

```
fun anyEven3() {
    val list = listOf(2, 3, 4, 6)
    val result = list.any{ item: Int -> item % 2 == 0 }
    println(result)
}
```

Lambda Expression: Example

Data type can be omitted if it can be inferred

```
fun anyEven4() {
    val list = listOf(2, 3, 4, 6)
    val result = list.any{ item -> item % 2 == 0 }
    println(result)
}
```

Lambda Expression: Example

For single arguments you can use the keyword **it**

```
fun anyEven5() {
    val list = listOf(2, 3, 4, 6)
    val result = list.any{ it % 2 == 0 }
    println(result)
}
```

Lambda Expression: Example

In multiline lambda, the last expression is the result

```
fun multiLineLambda():List<Int>{
    val list = listOf(2,3,4,5)
   return list.filter{
        it%2 == 0
        it % 4 == 0
    }
}
```

Lambda Expression: Example

Destructuring

```
fun destructure(){
    val map = mapOf(2 to "two",3 to "three",4 to "four",5 to "five")
    map.mapValues { (k, v) -> println("Key=$k, Value=$v") }
}
```

Lambda Expression: Example

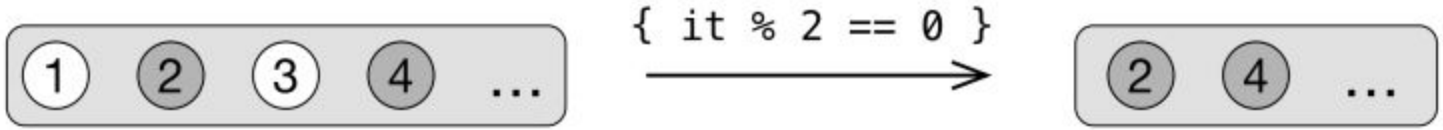
You can omit a parameter name if it is not used

```
fun mapValue(){
    val map = mapOf(2 to "two",3 to "three",4 to "four",5 to "five")
    map.mapValues { (_, v) -> println("Value=$v") }
}
```

```
fun mapValue(){
    val map = mapOf(2 to "two",3 to "three",4 to "four",5 to "five")
    map.mapValues { (k, _) -> println("Key=$k") }
}
```

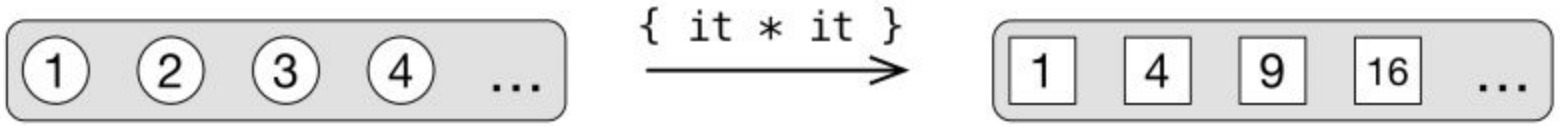
Operations on Collections

filter



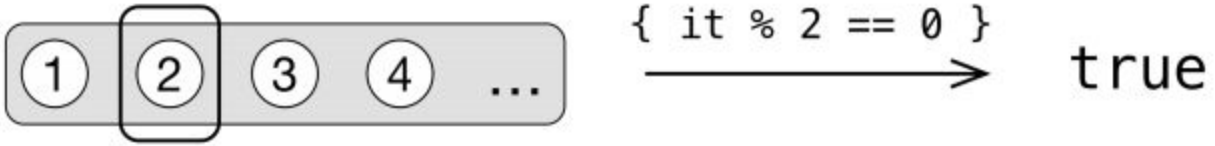
Operations on Collections

map



Operations on Collections

any (all, none)



Operations on Collections

find

Operations on Collections

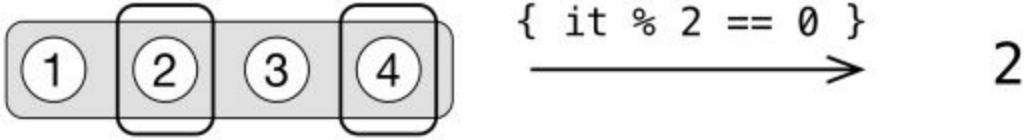
first / firstOrNull

Operations on Collections

first / firstOrNull

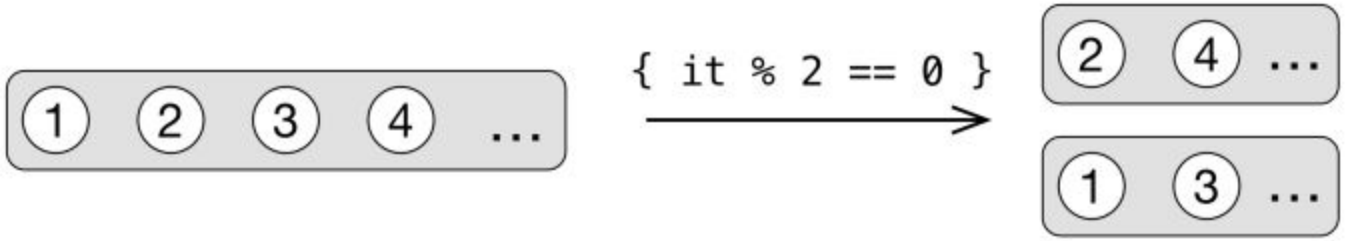
Operations on Collections

count



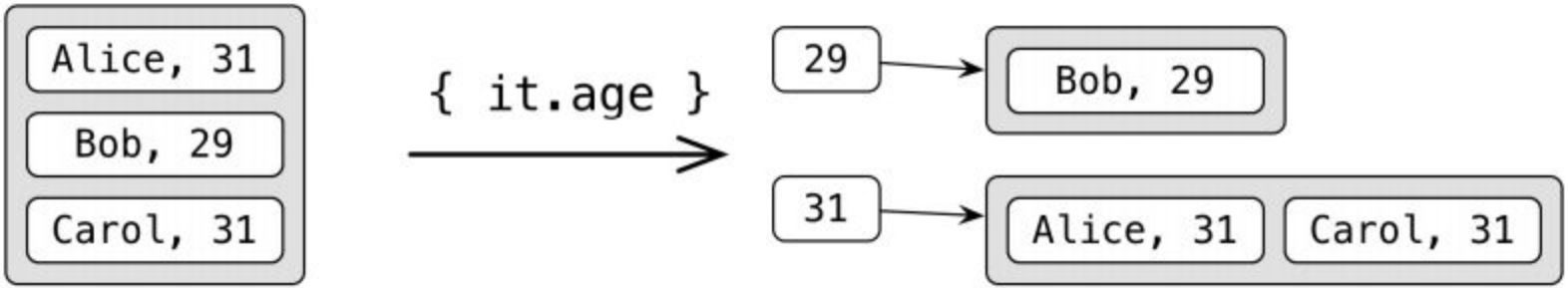
Operations on Collections

partition



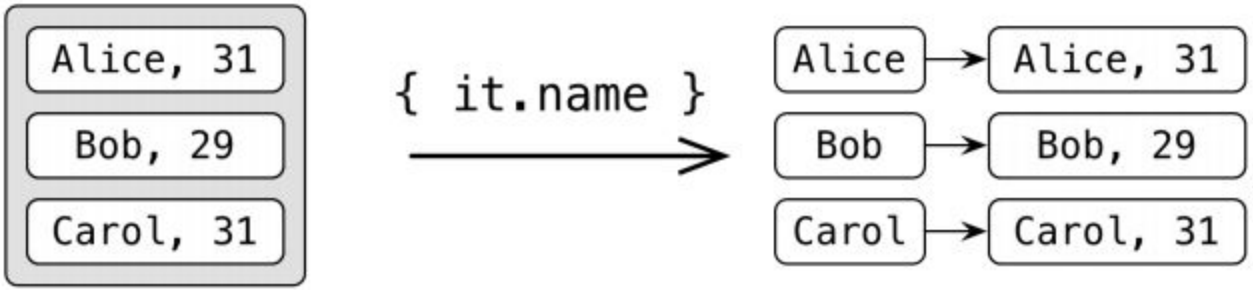
Operations on Collections

groupBy



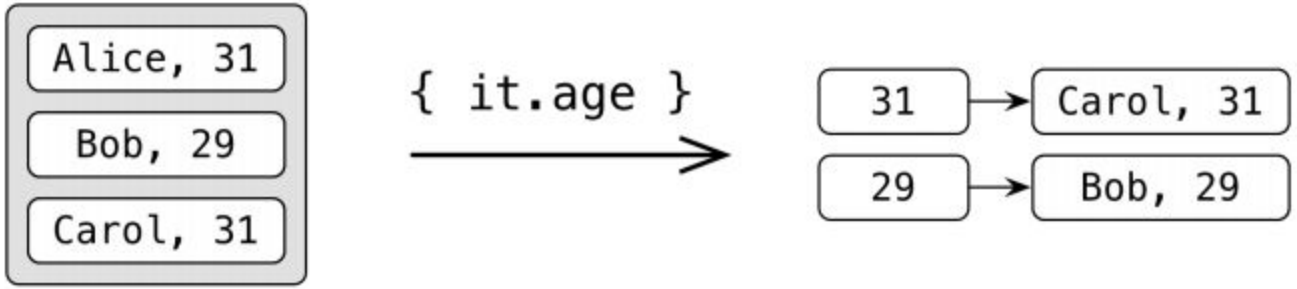
Operations on Collections

associateBy



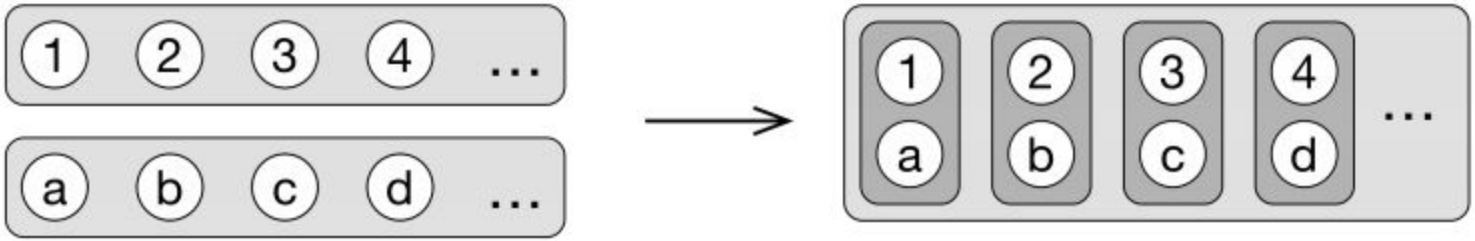
Operations on Collections

associateBy (removes duplicate)



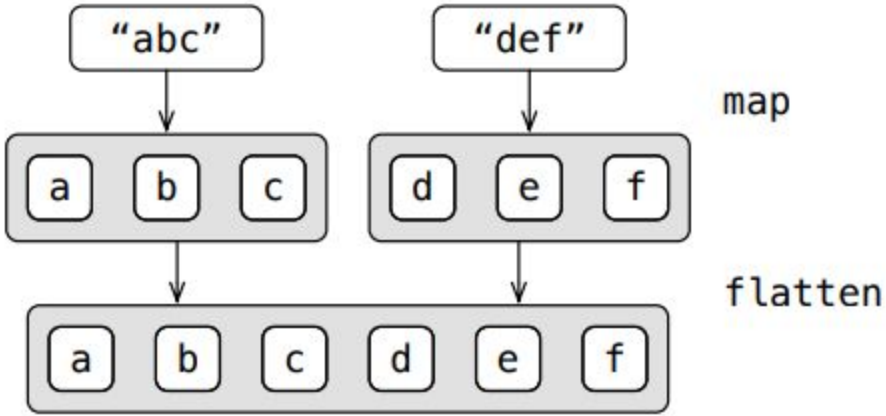
Operations on Collections

zip



Operations on Collections

flatMap

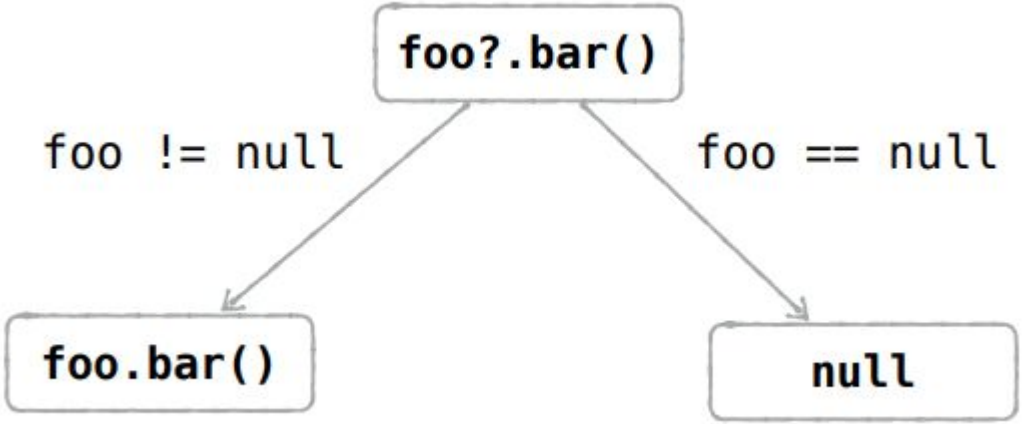


Nullability: Nullable types in Kotlin

Kotlin makes Null Pointer Exception a compile-time error (not runtime error)

```
val s1: String = null          ✗

val s2: String? = "can be null or non-null"

s1.length      ✓

s2.length      ✗
```

Nullability: Safe access



Nullability: Nullability operators

```
val s: String?

val length = if (s != null) s.length else null
```

```
val length = s?.length
```

Nullability: Nullability operators

```
val s: String?

val length: Int = if (s != null) s.length else 0
```

```
val length: Int = s?.length ?: 0
```

Safe casts: as

= instanceof

not necessary

```
if (any is String) {
    val s = any as String
    s.toUpperCase()
}
```

```
if (any is String) {
    any.toUpperCase()
}
```

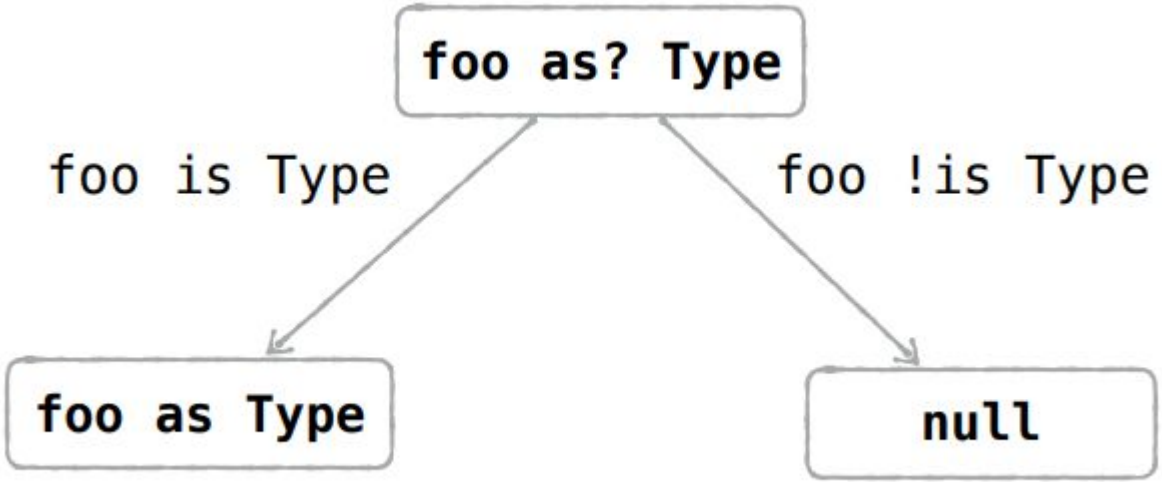
smart cast

Safe casts: as?

```
if (any is String) {
    any.toUpperCase()
}
```

```
(any as? String)?.toUpperCase()
```

Safe casts



Safe casts: as?

```
val s = if (a is String) a else null
```

```
val s: String? = any as? String
```

Standard Functions

apply

can be thought of as a configuration function

It allows you to call a series of functions on a receiver to configure it for use

```
import java.io.File

fun main() {
    // without apply
    val menuFile = File("menu-file.txt")
    menuFile.setReadable(true)
    menuFile.setWritable(true)
    menuFile.setExecutable(false)
}
```

```
import java.io.File

fun main() {
    // with apply
    val menuFile = File("menu-file.txt")
    menuFile.apply {
        setReadable(true)
        setReadable(true)
        setWritable(true)
        setExecutable(false)
    }
}
```

References

https://play.kotlinlang.org/koans/overview

https://kotlinlang.org/docs/reference/

https://www.coursera.org/learn/kotlin-for-java-developers

Jemerov, Dmitry, and Svetlana Isakova. Kotlin in action. Shelter Island, NY: Manning Publications, 2017

Skeen, Josh, and David Greenhalgh. Kotlin programming : the Big Nerd Ranch guide. Atlanta, GA: Big Nerd Ranch, 2018

https://www.youtube.com/user/JetBrainsTV