Kotlin Quickstart

Kotlin Feature Overview

- Maintained and developed by JetBrains
- Latest version: 2.2.10 (August 2025)
- Previous Major version: 1.9.25 (December 2023)
  - Version 1.9 is the last minor release prior to future release of Version 2.x
  - Android Studio Koala: Kotlin compiler default to 1.9.xx
- Official Online Documentation: http://kotlinlang.org
- Kotlin Multiplatform
  - Interoperate, Objective-C and Swift
  - CocoaPods Gradle Plugin
  - Xcode 14.1 + Kotlin Plugin

Group Exercise
Kotlin Online Tour

Nullable Types: Data May Be Absent

```kotlin
// Kotlin
var lastLoginDate: String? = null

fun smallestNumber(items: Array<Int>): Int? {
   // more code here
}
```

```python
// Python 3.5 (with typing hints)
from typing import Optional
lastLoginDate: Optional[str] = None

def smallestNumber(items: list[int]) -> Optional[int] :
   # more code here
```

Nullable Types: Practical Use Cases

- Nullable-Boolean: Result of an operation
  - **true** ⇒ operation succeeded
  - **false** ⇒ operation failed
  - **null** ⇒ operation has not completed
- Nullable-Date: Delivery date
  - "2023-08-17" ⇒ actual delivery date
  - **null** ⇒ item is not delivered yet
- Nullable-Float: Banking account balance
  - Positive ⇒ You have some money
  - Negative ⇒ You owe the bank some money
  - Zero ⇒ You have no money in the account
  - **null** ⇒ You don't have an account with this bank

Nullable Types & Safe Call Operator ?.

```
var one:String = "Tik"      // Non-nullable type can hold only Strings
print("We say ${one.toUpperCase()}")     // OUTPUT: We say TIK
```

```
// Nullable type: can hold either a String or null
var two:String? = "Tok"     // initialized to a String
var three:String? = null    // initialized to null

print("We say ${two?.toUpperCase()}")    // OUTPUT: We say TOK
print("We say ${three?.toUpperCase()}") // OUTPUT: We say null
print("We say ${two.toUpperCase()}")     // ERROR: Only safe call is allowed
print("We say ${three.toUpperCase()}")   // ERROR: Only safe call is allowed
```

Nullable Types & “Elvis” Operator ?:

```
var two:String? = “Tok”    // Nullable type: can hold either a String or null
var three:String? = null   // Nullable type: can hold either a String or null

// Elvis op implies “if-else”
val dos = two ?: “Flip”         // dos is a String (non-nullable)
val tres = three ?: “Flop” // tres is a String (non-nullable)

print(“We say ${two?.toUpperCase()}”)    // OUTPUT: We say TOK
// The following calls do NOT require safe call operator
print(“We say ${dos.toUpperCase()}”)     // OUTPUT: We say TOK
print(“We say ${tres.toUpperCase()}”)    // OUTPUT: We say FLOP
```


Non-null Assertion operator (!!)

```
var two:String? = “Tok”    // Nullable type: can hold either a String or null
var three:String? = null   // Nullable type: can hold either a String or null

print(“We say ${two!!.toUpperCase()}”)   // OUTPUT: We say TOK
print(“We say ${three?.toUpperCase()}”)  // Output null (safe call)
print(“We say ${three!!.toUpperCase()}”) // Runtime crashed
```

Non-null Assertion operator (!!)

```
var two:String? = “Tok”    // Nullable type: can hold either a String or null
var three:String? = null   // Nullable type: can hold either a String or null

fun greet (who: String) {
  print (“Hello $who”)
}

// Use cases when !! is required
greet(two)        // Compile error due to type mismatch
greet(two!!)      // OK
greet(three!!)    // Compile OK but Null-Pointer Exception at runtime
```


Review of Nullable Types

Why use nullable types?

?:

- What is this operator called?
- Why do we need to use it?

?.

- What is this operator called?
- Why do we need to use it?

!!

- What is this operator called?
- Why do we need to use it?

Functions

Function: Parameters, Arguments, Invocation

- (Input) Parameters: placeholder variables used for receiving actual values at runtime
- Arguments: actual values supplied into a function during invocation

```
// Year is a PARAMETER
fun isLeap(year: Int): Boolean {
  // more code here
}
```

```
// Invocation: 1900 is an ARGUMENT
if (isLeap(1900)) {
  // more code here
}

// Can also be called with named param
if (isLeap(year = 1900)) {
  // more code here
}
```

Default Parameter Values

```
fun parseInteger(str: String, base: Int = 10): Int {
  // more code here
}
```

```
val num1 = parseInteger("287")                // parse as decimal
val hex1 = parseInteger("4AD0", base = 16)    // parse as hexadecimal
val bin1 = parseInteger("100101", base = 2)   // parse as binary
```

Playground

Statements vs. Expressions

- Java control structures are **statements**
- Most Kotlin control structures (other than loops) are **expressions**

```
fun maxOfTwo(a: Int, b: Int): Int {
  if (a > b) return a
  else return b
}
```

```
fun maxOfTwo(a: Int, b: Int): Int {
  return if (a > b) a else b
}
```

Single Return Functions

- When the function body is *a single return expression*, the body can be **replaced with the expression itself**

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

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

```
fun maxOfTwo(a: Int, b: Int): Int {
  return if (a > b) a else b
}
```

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


Extension Functions

Extension Functions

- A **concise** way of ***adding*** *new behaviors* (methods/functions) to an existing type
  - The existing type can be either Kotlin **builtin** type/class or **user-defined** types/class
  - **Can't** use extension functions to ***override*** *existing behaviors*
- Compared to Java
  - Can only extend a class (cannot add new behaviour to other types)
  - Can override existing behaviors
  - Requires use of class inheritance

Extension to existing type

```
// Kotlin
fun Int.isPowerOf(base: Int): Boolean {
   var x = this
   while (x > 1 && x % base == 0) {
      x = x / base
   }
   return x == 1
}

if (27.isPowerOf(2)) {

}

if (27.isPowerOf(3)) {

}
```

Which one: Extension or Ordinary?

```
// Extension Function on Int
fun Int.isPowerOf(base: Int): Boolean {
   var x = this
   while (x > 1 && x % base == 0) {
      x = x / base
   }
   return x == 1
}

if (27.isPowerOf(2)) {

}

if (27.isPowerOf(3)) {

}
```

```
// Ordinary Function
fun isPowOf(value: Int, base: Int): Boolean {
   var x = value
   while (x > 1 && x % base == 0) {
      x = x / base
   }
   return x == 1
}

if (isPowOf(27, 2)) {

}

if (isPowOf(27, 3)) {

}
```


Which one: Extension or Ordinary?

- The functionality of an extension function on *builtin types* **can be substituted** with an ordinary function. But using extension functions may:
  - Improve code readability: similar coding style to that used by the Kotlin stdlib
  - Code/File organization:
    - Avoid creating many “loose” top-level functions throughout your code
    - All extension functions on `String` can be written in a separate file `my-string-extensions.kt`
- Extension functions on a *class type* open the possibility to add new features, even to “closed” classes (cannot be inherited to child classes)

Control Structures

Control Structures

- Expressions (have side effects and value)
  - if-else expressions
  - when expression
- Statements (have side effects but no value)
  - while
  - do-while
  - for-loops

`if` expression

```
var max = a
if (a < b) max = b
```

```
max = if (a < b) b else a
```

```
if (a < b)
    max = b
else
    max = a
```

```
max = if (a < b) {
  println("b is bigger")
  b * 5   // last expression in the block
          // becomes the value of the block
} else {
  println("a is bigger")
  a + 10
}
```


`when` expression

```
var dayOfWeek: Int

when (dayOfWeek) {
  0 -> println("Scenic Sunday")
  1 -> println("Marvelous Monday")
  2 -> println("Tranquil Tuesday")
  3 -> println("Witty Wednesday")
  4 -> println("Thoughtful Thursday")
  5 -> println("Fabulous Friday")
  6 -> println("Serene Saturday")
  // This else below is required
  else -> println("Woozy Weekday")
}
```

```
var dayOfWeek: Int
var msg:String
msg = when (dayOfWeek) {
  0 -> "Scenic Sunday"
  1 -> "Marvelous Monday"
  2 -> "Tranquil Tuesday"
  3 -> "Witty Wednesday"
  4 -> "Thoughtful Thursday"
  5 -> "Fabulous Friday")
  6 -> "Serene Saturday"
  // else is required
  else -> "Woozy Weekday"
}

println(msg)
```

```
enum class WorkDay {
  MON, TUE, WED, THU, FRI
}
val dow:WorkDay = WorkDay.TUE
var msg:String
msg = when (dow) {
    WorkDay.MON -> "Moonday"
    WorkDay.TUE -> "Twosday"
    WorkDay.WED -> "Whensday"
    WorkDay.THU -> "Trustday"
    WorkDay.FRI -> "Friedday"
    // No else required
}
```

- *"Case labels" must be exhaustive*
- *No break required in between "cases"*

For-in

```
// 3, 4, ..., 11
for (d in 3..11) {
  print(d)
}
```

```
// 3, 4, ..., 10
for (d in 3 until 11) {
  print(d)
}
```

```
for (d in 10 downTo 0 step 2) {
  println("Count down $d")
}
```

```
val arr = listOf(23, 37, 71)
for (n in arr) {
  println(n)
}
```

```
val msg = “Hello world”
for (c in msg) {
  println(n)
}
```

Playground

For-in and Maps (Dictionaries)

```
// Kotlin
val httpCode = mapOf(200 to "OK",
                     201 to “Created”,
                     404 to "Not Found")
for ((k,v) in httpCode) {
  println("HTTP status $k means $v")
}
```

```
// Python
httpCode = {200: "OK",
            201: “Created”,
            404: "Not Found"}
for k,v in httpCode.items():
  print(f"HTTP status {k} means {v}")
```

Classes

Classes

City.java

```
public class City {
  private String name;
  private int population;

  public City(String n, int p) {
    this.name = n;
    this.population = p;
  }
  public String getName() {
    return this.name;
  }
  public void setPopulation(int pop) {
    this.population = pop
  }
}
```

City.kt

```
class City (
  val name: String
  var population: Int)
```

City.kt

```
data class City (
  val name: String;
  var population: Int)
// equals, toString(), hashCode() will
// be automatically generated
```

```
val ada = City("Ada", 15000)
println("Population ${ada.population})
ada.population = 15622
ada.name = "Alma" // ERROR name is immutable
```

Playground

Classes

City.py

```
class City:
  def __init__ (self, name, population):
    self.name = name
    self.population = population
```

main.py

```
ada = City("Ada", 15000)
ada.population = 15622
ada.name = "Alma"
```

City.kt

```
class City (
  val name: String
  var population: Int)
```

City.kt

```
data class City (
  val name: String;
  var population: Int)
// equals, toString(), hashCode() will
// be automatically generated
```

```
val ada = City("Ada", 15000)
println("Population ${ada.population})
ada.population = 15622
ada.name = "Alma" // ERROR name is immutable
```

Playground

Class Primary Constructor & Properties

Java (instance variables + getter + setter) becomes Kotlin properties

```
class City constructor (
  val name: String
  var population: Int,
  private var mayorSalary: Int)
```

constructor keyword is optional

```
class City (
  val name: String
  var population: Int,
  private var mayorSalary: Int)
```

In the above example:

- `name` *is a read-only property (Kotlin generates a field and a getter)*
- `population` *is a writable property (Kotlin generates a field, a getter, and a setter)*
- `mayorSalary` is a writable but private property

Primary constructor & initializer block

ChessGame.kt

```
enum class PieceType {
  PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING
}
enum class PieceColor { BLACK, WHITE }

class ChessPiece (
  val color:PieceColor,
  val type:PieceType = PieceType.PAWN,
  var row:Int = 1, var column:Int = 1) {

  init { // run after the primary constructor
    if (color == PieceColor.WHITE)
       row = if (type == PieceType.PAWN) 2 else 1
    else
       row = if (type == PieceType.PAWN) 7 else 8
  }

  fun isValidMove(toRow:Int, toCol:Int): Boolean {
    // more code here
  }
}
```

ChessGame.java

```
class ChessPiece {
  PieceColor color;
  PieceType type;
  int row, column;

  public ChessPiece(PieceColor bw, PieceType t, int r, int c) {
    color = bw;
    type = t;
    column = c;

    if (color =- PieceColor.WHITE)
       row = type == PieceType.PAWN ? 2 : 1
    else
       row = type == PieceType.PAWN ? 7 : 8
  }
  public boolean isValidMove(int toRow, int toCol) {
    // more code here
  }
}
```

Playground

Secondary constructors

primary constructor

init block (goes together with primary constructor)

secondary constructor

ChessGame.kt

```
class ChessPiece (val color:PieceColor,
  val type:PieceType = PieceType.PAWN,
  var row:Int = 1, var column:Int = 1) {

  init { // run after the primary constructor
    if (color == PieceColor.WHITE)
       row = if (type == PieceType.PAWN) 2 else 1
    else
       row = if (type == PieceType.PAWN) 7 else 8
  }

  // secondary constructor
  constructor(pawnColor: PieceColor, atColumn: Int):
      this(color = pawnColor, column = atColumn)
  {
    println("Creating a $pawnColor pawn at column $atColumn")
  }
}
```

must invoke the primary constructor

Playground

Class Inheritance: open or final

```
// Java
class MyClass {

}

// Inheritance is allowed
class AnotherClass: MyClass {
}
```

```
// Kotlin (default to final class)
class YourClass {

}

// Inheritance is NOT allowed
class YerClass: YourClass() {
}
```

```
// Java
final class MyKlazz {

}

// ERROR: Inheritance is NOT allowed
class AnotherClass: MyKlazz {
}
```

```
// Kotlin
open class OkKlazz {

}

// Inheritance from open class is allowed
class YerClass: OkKlazz() {
}
```


Extension Functions on a Class

```
class City (
  val name: String
  var population: Int,
  private var mayorSalary: Int)
```

```
fun City.prettify(prefix: String, suffix: String) {
  return “$prefix ${this.name} has “ +
     “${this.population} people $suffix”
}
```

OK

```
val grp = City(“Grand Rapids”, 87345, 125000)

println(grp.prettify(prefix = “[“, suffix = “!”))
```

```
// This extension function WON’T compile
fun City.averageMayorSalaryPerPopulation(): Double {
  return this.mayorSalary.toDouble()
      / this.population.toDouble()
}
```

Can’t access private variables

Collections

Functions for Creating Collections

	Construct from elements	Build from elements

ArrayList	`val fruits = listOf("Apple", "Banana", "Cherry")`
	`val primes = mutableListOf(3, 5, 7, 11, 13)`
	`primes.add(17)`
	
	`// Data type is required`
	`val emptyList = mutableListOf<Float>()`
		`val fruits = buildList {`
		`  add("Apple")`
		`  add("Banana")`
		`  add("Cherry")`
		`}`

Set	`val fruits = setOf("Apple", "Banana", "Apple")`
	`val emptySet = mutableSetOf<Int>()`
	`print(fruits.size)  // output: 2`
		`val fruits = buildSet {`
		`  add("Apple")`
		`  add("Banana")`
		`  add("Cherry")`
		`}`

Map	`val monthLen = mapOf("Jan" to 31, "Feb" to 28)`
	`val romanNums = mutableMapOf<Char,Int>()`
	`romanNums['I'] = 1`
	`romanNums['V'] = 5`
		`val monthLen = buildMap {`
		`  put("Jan", 31)`
		`  put("Feb, 28)`
		`}`

Playground

More Collection Functions (1 of 2)

Category	Extension Functions
Transformation	N to N: `map, mapIndexed, mapKeys, mapValues` 2N to N: `zip, unzip` More to Few: `associate, associateWith, associateBy` Few to More: `flatten, flatMap`
Filtering	More to few: `filter, filterIndexed, filterNot, filterNotNull, filterIsInstance` N+M to (N, M): `partition` N to boolean: `any, none, all, contains, isEmpty, isNotEmpty`
Grouping	More to Few: groupBy, groupingBy
Subcollection	`slice(), take(N), takeLast(N), drop(N), dropLast(N), takeWhile, takeLastWhile, dropWhile, dropLastWhile, chunked(N), zipWithNext`
Get one	`first(), last(), elementAt(), elementAtOrNull, find, findLast`

More Collection Functions (2 of 2)

Category	Extension Functions
Ordering	`sorted, sortedDescending, sortedBy, sortedByDescending, sortedWith, reversed, shuffled`
Aggregate	`minOrNull, minByOrNull, minWithOrNull, minOfOrNull, minOfWithOrNull, maxOrNull, maxByOrNull, maxWithOrNull, maxOfOrNull, maxOfWithOrNull sumOf, count,`
Fold & reduce	Start from the first element: `fold, foldIndexed` Use the first element as initial value, start from the second element: `reduce, reduceIndexed, reduceOrNull, reduceIndexedOrNull` Apply in reverse order: `foldRight, foldRightIndexed, reduceRight, reduceRightIndexed, reduceRightOrNull, reduceRightIndexedOrNull`