رواد مصر الرقمية

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

Kotlin for Java Developers

Presented by: Eng. Ahmed Atef
Mobile Instructor

7/24/2025

Why Kotlin?

- No more semicolons
- Easier than Java
- More flexible
- Helps you to avoid null pointer exception
- Interoperable with Java
- Shorter lines of code
- Line is more concise
- Increased stability for Google Home App by 30%
- The existence of Kotlin Multiplatform (KMP)

Hello World!

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

```
Process finished with exit code 0
```

means → Program has ended successfully

Variables

Data Type	Value Stored	Weight in Memory
Int	-2,147,483,648 → 2,147,483,647	4 bytes
Double	Decimal Digits (15-16)	8 bytes
String	“Candroid”	
Boolean	true or false	1 byte
Char	‘A’	2 bytes
Byte	-128 → 127	1 byte
Short	-32,768 → 32,767	2 bytes
Long	-2^63 → 2^63-1	8 bytes
Float	Decimal Digits (6-7)	4 bytes

Var (Variable) & Val (Value)

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

    var x = 50
    x = 100
    x = 101
    val y = 50
    //Val cannot be reassigned
    y = 100

}
```

Note: **val** can be assigned a value only once

Type Inference

- Kotlin is a statically-typed language, which means that every variable, every expression has a type, even if you omit the type, the compiler infers it for you.

Example

```
val name = "Candroid"
```

Behind the scenes

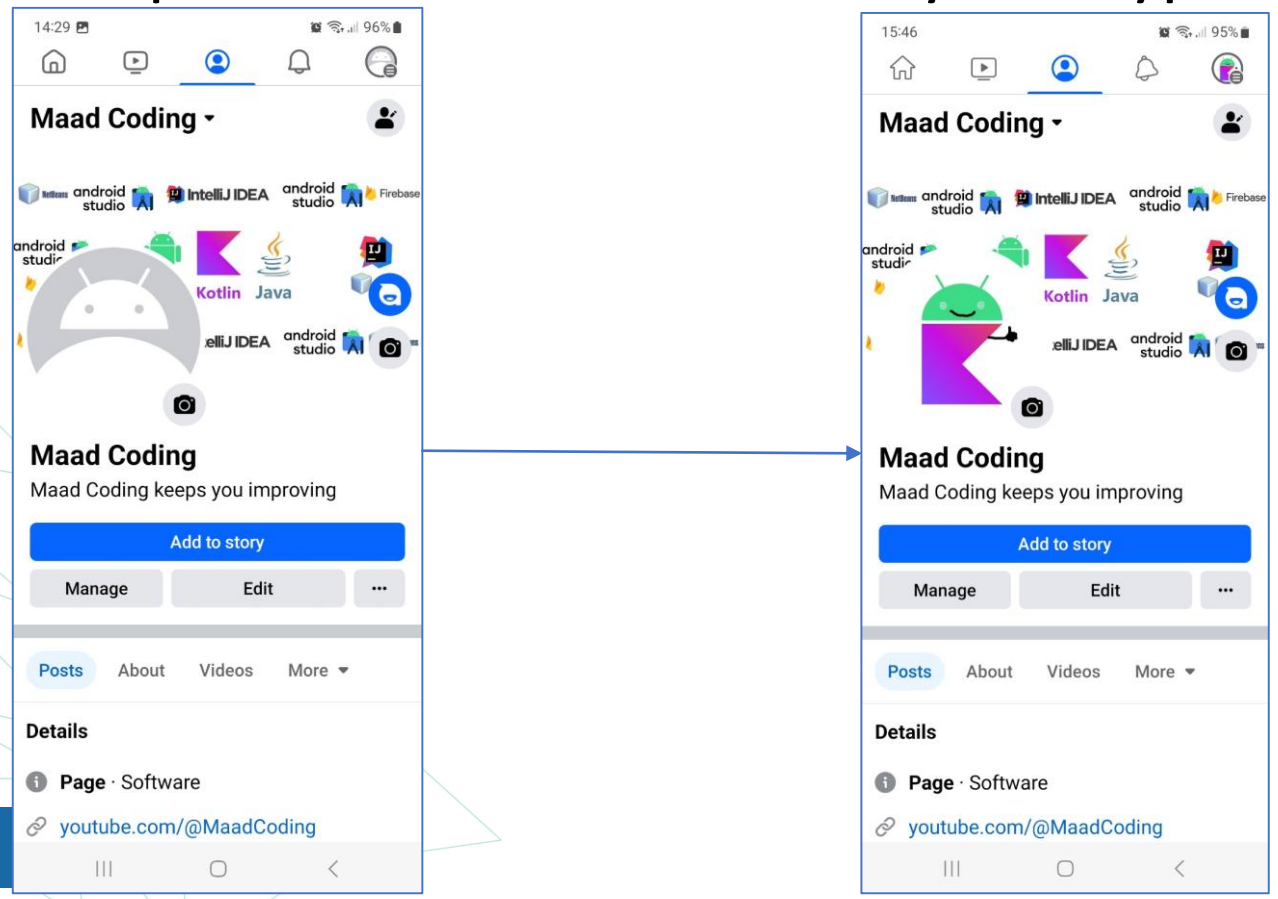
```
val name: String = "Candroid"
```

Constants

```
const val NAME = "Kotlin"
```

Null

- null is a keyword used to represent anything (object) that doesn't exist
- By default, you cannot pass null as a value for any datatype like Int



Nullability

- Kotlin helps avoid null pointer exceptions (NPE)
- Use the question mark operator to indicate that a variable can be null

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

    var x: Int? = null
    var name: String? = null
    println(name)
    name = "Candroid"
    println(name)

}
```

I Love NPE!!

The not-null assertion operator (!!) converts any value to a non-null type and throws an exception if the value is null.

Try to avoid it unless you are sure that the variable is not null

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

    val b: String? = null
    println(b!!.length)

}
```

Dealing with Nullable Types

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

    //name can hold a "value" or "null"
    var name:String? = "Candroid"
    println(name?.length) //8

    name = null
    println(name) //null

}
```

Note: (?.) is called a safe call operator

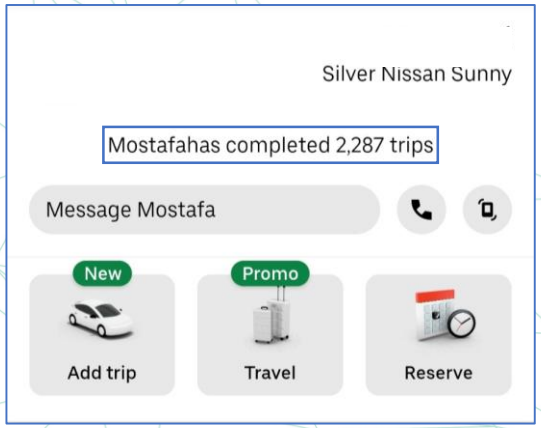
Elvis Operator ?:

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

    val name:String? = null
    val length =
        if (name != null) {name.length}
        else {0}
    println(length)

    val name2:String? = null
    val length2 = name2?.length ?: 0
    println(length2)

}
```

Concatenation (+) “Java & Kotlin”

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

    val name = "Candroid"
    val age = "2"

    println(name)
    println(age)
    println("My name is " + name)
    println("My age is " + age)
    println("My name is " + name + " and my age is " + age)

}
```

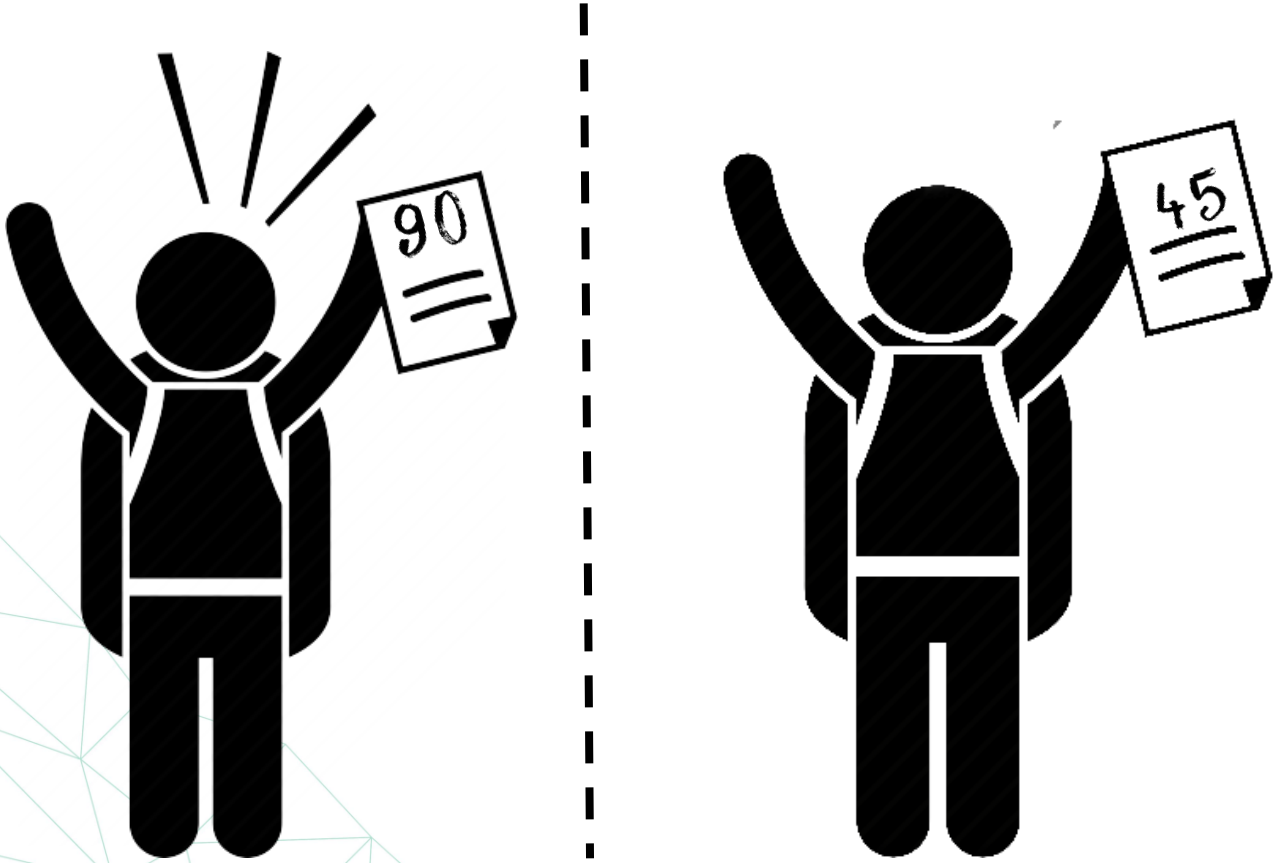
String Templates ($) “Kotlin”




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

    val captainName = "Mostafa"
    // 'd' represents an integer value
    val totalTrips = "%,d".format( ...args: 2287)

    println("$captainName has completed $totalTrips trips")

}
```


https://kotlinlang.org/docs/strings.html#string-templates

Conditions: when expression



https://kotlinlang.org/docs/control-flow.html#when-expression

Conditions: when expression

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

    val grade = 92
    when (grade) {
        90, 91, 92 -> {
            println("Congratulations!")
            println("A+")
        }

        80 -> { println("B+") }

        else -> { println("Invalid") }
    }

}
```

Conditions: when expression

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

    val grade = 54
    when (grade) {
        in 50 ≤ .. ≤ 100 -> println("Pass")

        !in 50 ≤ .. ≤ 100 -> println("Fail")
    }

}
```

When Expression (Training)

- Write a program that checks for a character if it ranges from
  - **‘a’ to ‘z’** or ranges from **‘A’ to ‘Z’**, then print “Letter is found”
  - If the character ranges from **‘0’ to ‘9’**, then print “Number is found”
  - Otherwise print “Special character is found”
- Your character value is **‘M’**

When Expression (Note)

```
when {
    x < 0 -> print("x < 0")
    x > 0 -> print("x > 0")
    else -> {
        print("x == 0")
    }
}
```

The condition can be inside of the branches.

Solution

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

    val letter = 'M'

    when (letter) {
        in 'a'..'z', in 'A'..'Z' -> println("Letter is found")
        in '0'..'9' -> println("Number is found")
        else -> println("Special character is found")
    }

}
```

Loops: for (Training 1)

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

    for (i: Int in 1..4){
        println(i)
    }

}
```

Loops: for (Training 2)

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

    for (m in 20 downTo 0 step 5){
        println(m)
    }

}
```

Loops: for (Training 3)

Write a program that prints this sequence using for loop:

1, 5, 9, 13, 17, 21…101

Loops

There are break and continue labels for loops:

```
myLabel@ for (item in items) {
    for (anotherItem in otherItems) {
        if (...) break@myLabel
        else continue@myLabel
    }
}
```

Solution

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

    for (i in 1..101 step 4){
        println(i)
    }

}
```

Conditions: Training

- Write a program that prints the largest number out of three whole numbers stored in variables using if expression
- Let variable “a” = 5
- Let variable “b” = 2
- Let variable “c” = 4

Conditions: Solution

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

    val a = 5
    val b = 2
    val c = 4
    val largest =
    if (a >= b && a >= c) {a}
    else if (b > c) {b}
    else {c}
    println("Largest number is $largest")

}
```

Array Definition

- Collection of elements with the same datatype
- Each index is a pointer to a specific element inside the array
- Arrays use zero-based indexing
- Kotlin also has classes that represent arrays of primitive types without boxing overhead: ByteArray, ShortArray, IntArray, and so on.

```
//Index                              0   1   2   3   4
val numbers: Array<Int> = arrayOf(10, 60, 40, 20, 15)
```

```
numbers[4]
```

This is called subscript syntax

Printing Array's Data

```
for (i in 0≤ until <numbers.size)
    println(numbers[i])
```

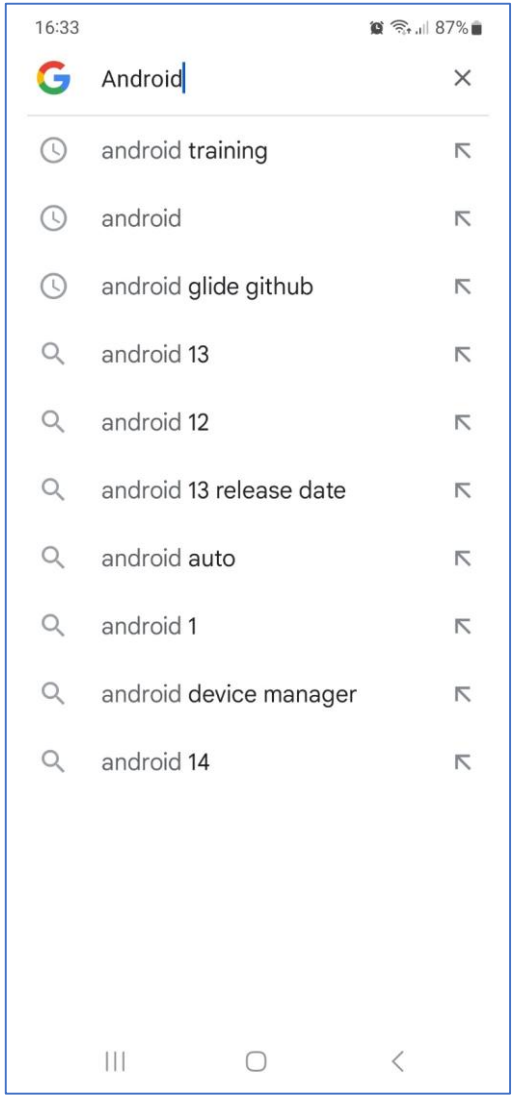
You can use element, item, or the singular form of the array name (number) when printing

```
for (element in numbers)
    println(element)
```

Loop over elements without needing its index

Task

- Google has an array of Strings with 100 elements.
- Write a program that will print the elements that contain the word Android, because the user wants to search for it
- You can check this GitHub Gist to see the array. It contains 10 elements with the word Android



Function Syntax

fun functionName() : returnType{}

function saveData() Int/Double/Char...{}

addUser() Unit{}

Note: Unit is like void in Java

Function (Training 1)

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

fun sum(): Unit{
    println(5+6)
}
```


https://kotlinlang.org/docs/functions.html#unit-returning-functions

Function (Training 2)

- Note: Functions that take other functions as arguments are called higher order functions.

```
fun main(args: Array<String>) {
    val result = makeMobile( noOfCameras: 3,  ram: 8,  storage: 128,  size: 6.4)
    println(result * 2)
}

fun makeMobile(noOfCameras: Int, ram: Int, storage: Int, size: Double): Double {
    val price = (noOfCameras * 300) + (ram * 500) + (storage * 10) + (size * 50.5)
    return price
}
```

Functions: Default Values

```
fun main(args: Array<String>) {
    repeatChar()
    repeatChar( char: '#')
    //repeatChar(5) --> Error
    repeatChar( char: '#',  repeatNumber: 5)
}

fun repeatChar(char: Char = '*', repeatNumber: Int = 10){
    repeat(repeatNumber){  it: Int
        print("$char ")
    }
}
```

Default values = Default value for the parameter

Functions: Named Arguments

```
fun main(args: Array<String>) {
    repeatChar()
    repeatChar( char: '#')
    repeatChar(repeatNumber = 5)
    repeatChar( char: '#',  repeatNumber: 5)
}

fun repeatChar(char: Char = '*', repeatNumber: Int = 10){
    repeat(repeatNumber){  it: Int
        print("$char ")
    }
}
```

Functions: Named Arguments

- When using named arguments in a function call, you can freely change the order they are listed in
- Reordering the arguments using named arguments will not affect the output
- To achieve this in Java, we used overloading methods
- Overloading: Functions with the same name and within the same class, but with different parameters:
  - Number of Parameters
  - Datatype
  - Arrangement

Local Functions

```
fun main() {
    val x = 6.0
    val y = 4.0
    doArithmeticOperations(x, y)
}

fun doArithmeticOperations(n1: Double, n2: Double) {
    fun sum() { println(n1 + n2) }
    fun minus() { println(n1 - n2) }
    fun multiply() { println(n1 * n2) }
    fun modulus() { println(n1 % n2) }
    fun divide() { println(n1 / n2) }
    sum(); minus(); multiply(); modulus(); divide()
}
```

Training

Send the numbers “7” & “8” as parameters to a function that returns the largest number among them, then print the result to the user in your main function.

Solution 1

```
fun main(args: Array<String>) {
    val result = max( a: 7, b: 8)
    println(result)
}

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

Solution 2

```
fun main(args: Array<String>) {
    val result = max( a: 7, b: 8)
    println(result)
}

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

Solution 3

```
fun main(args: Array<String>) {
    println(max( a: 7, b: 8))
}

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

Single-Expression Function

You can write your function as a Single-Expression Function if:

- The function has a return type
- The function body contains one line only

Example

```
fun getYear(): Int{
    return Calendar.getInstance().get(Calendar.YEAR)
}
```

```
fun getYear() = Calendar.getInstance().get(Calendar.YEAR)
```

Try & Catch Note

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

    val isSuccess = try {
        /*
        Getting some pictures from a server
        with no internet connection, or host name doesn't exist
         */
        true
    }
    catch (e: UnknownHostException) {
        //Catch the exception that happened
        false
    }
    finally {
        /*
        Close the server connection whether the request
        succeeded or failed
         */
    }

}
```

Try is also an expression, which means it can have a return value

رواد مصر الرقمية

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

Try & Catch VS If

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

    try {
        /*
        Getting some pictures from a server
        with no internet connection,
        or host name doesn't exist
         */
    }
    catch (e: UnknownHostException) {
        //Catch the exception that happened
    }

}
```

```
Process finished with exit code 0
```

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

    val hasConnection = true

    if (hasConnection) { //Condition is required
        /*
        Getting some pictures from a server
        with no internet connection,
        or host name doesn't exist (Unknown Host "NULL")
         */
    }

}
```

```
Exception in thread "main" java.net.UnknownHostException
    at MainKt.main(Main.kt:8)

Process finished with exit code 1
```

Try & Catch (Note)

Do not use exceptions for:

- Control flow
- Manageable errors

Setters & Getters

- Class properties (variables) contains implicit setters and getters by default
- Don’t write setters and getters explicitly unless you need to write extra code
- A field identifier is only used as a part of a property to hold its value in memory

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

    val user1 = User()
    user1.name = "Kotlin"
    user1.email = "MaadKotlinCoder@gmail.com"
    user1.password = "Hello123"

    println(user1.password)

}
```

```
class User {

    var name= ""
    var email = ""

    var password = ""
        set(value) {
            field = value + ('a'..'z').random()
            println("Saved password: $field")
        }
        get() = field.dropLast(1)

}
```

Task

Convert this program to use setters & getters instead of repeating the code.

```
class Student {

    var age = 0

}
```

```
fun main(args: Array<String>) {
    val s1 = Student()
    s1.age = 2
    if (s1.age < 5)
        println("Age is not valid")
    else
        println("Age saved successfully")

    val s2 = Student()
    s2.age = 6
    if (s2.age < 5)
        println("Age is not valid")
    else
        println("Age saved successfully")
}
```

Constructor

```
class Student {

    var name = ""
    var age = 0

    constructor(name: String, age: Int){
        this.name = name
        this.age = age
    }

}
```

Inside main fun

```
val s1 = Student( name: "Candroid",  age: 8)
println(s1.name)
println(s1.age)
```

Simplifying our Code

```
class Student constructor(name: String, age: Int) {

    var name = name
    var age = age

}
```

Inside main fun

```
val s1 = Student( name: "Candroid",  age: 8)
println(s1.name)
println(s1.age)
```

Simplifying our Code (Again)

By using var or val before any parameter in the constructor, this creates the member properties for us

```
class Student constructor(val name: String, val age: Int) {


}
```

Inside main fun

```
val s1 = Student( name: "Candroid",  age: 8)
println(s1.name)
println(s1.age)
```

The Last Simplify

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

Inside main fun

```
val s1 = Student( name: "Candroid",  age: 8)
println(s1.name)
println(s1.age)
```

Primary Constructor Summary

- Special function used to initialize a newly created object
- Starts with the class name
- Doesn’t have a return type even Unit
- A class can have only one primary constructor
- Any class contains a default empty constructor with no parameters called:
  - Default Constructor = It exists without writing it
  - Empty Constructor = Doesn’t have a body { }
  - Non-parameterized constructor = Has Zero parameters
  - No-argument constructor = No data is sent to it

Note: To run code during object creation, you can use init{} inside the class body

Secondary Constructors

Unlike a primary constructor, a class can contain multiple secondary constructors, but each secondary constructor needs to initialize the primary constructor

```
class Product(val name: String, val price: Double) {

    constructor(name: String, price: Double, expiryYear: Int) : this(name, price) {
        //Check expiry date value and make discounts if needed
    }

}
```

Inside main fun

```
val p1 = Product( name: "AL SA'A RICE",  price: 39.99)
val p2 = Product( name: "AHMAD TEA",  price: 62.5,  expiryYear: 2027)
```

Object Note

The order of initialization:

the primary constructor -> the init block -> the secondary constructor

Kotlin Note

Any declaration in Kotlin is public and final by default. This includes:

- Variables
- Methods
- Classes

Inheritance

```
//Class is "open" for inheritance
//Known as: Parent - Super - Base class
open class Human {
    var name = ""
    var age = 0
    var address = ""
}
```

```
//Known as: Child - Sub - Derived class
class Baby: Human() {
    var toy = ""
}
```

```
fun main(args: Array<String>) {
    val b1 = Baby()
    b1.name = "Koty"
    b1.age = 2
    b1.toy = "Teddy Bear"
    b1.address = "Babisland"
    println("${b1.age}-year-old ${b1.name} from ${b1.address} loves his ${b1.toy}")
}
```

A child class is also called **inheritor**

Inheritance Notes

- Kotlin is single in inheritance which means that it can inherit from one single class at a time
- The parent class for all classes in Kotlin is Any
- Inheritance represents the “IS-A” relationship between classes
  - A baby is a human
  - A circle is a closed shape
  - A Lemon tree is a specific type of plant

Methods Overriding

- Inside of a class, functions are referred to as methods.
- You can prohibit further overriding by marking a member final.

```
open class Movie {
    open fun famousQuote() {
        println("Quote is ")
    }
}
```

```
class LionKing : Movie() {
    override fun famousQuote() {
        super.famousQuote()
        println("Hakuna Matata")
    }
}
```

```
fun main(args: Array<String>) {
    val lion = LionKing()
    lion.famousQuote()
}
```

Self Study: https://kotlinlang.org/docs/inheritance.html#overriding-properties

Property Overriding

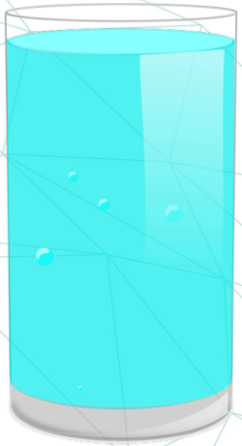
```
open class Student {
    open val age = 5
}
```

→

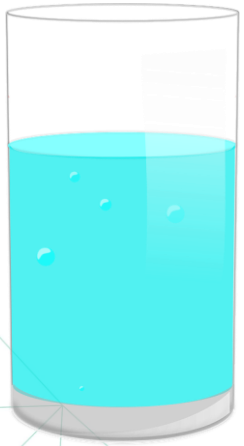
```
class CollegeStudent : Student() {
    override val age = 18
}
```

Abstraction Notes

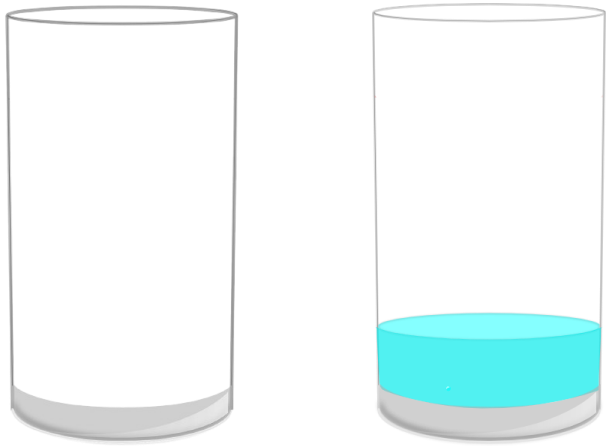
- Use a colon (:) instead of using extends and implements
- An abstract class has a constructor, but an interface has no constructor
- Use an interface if you have a lot of abstract methods and little default implementations like one method or two with a method body



Class



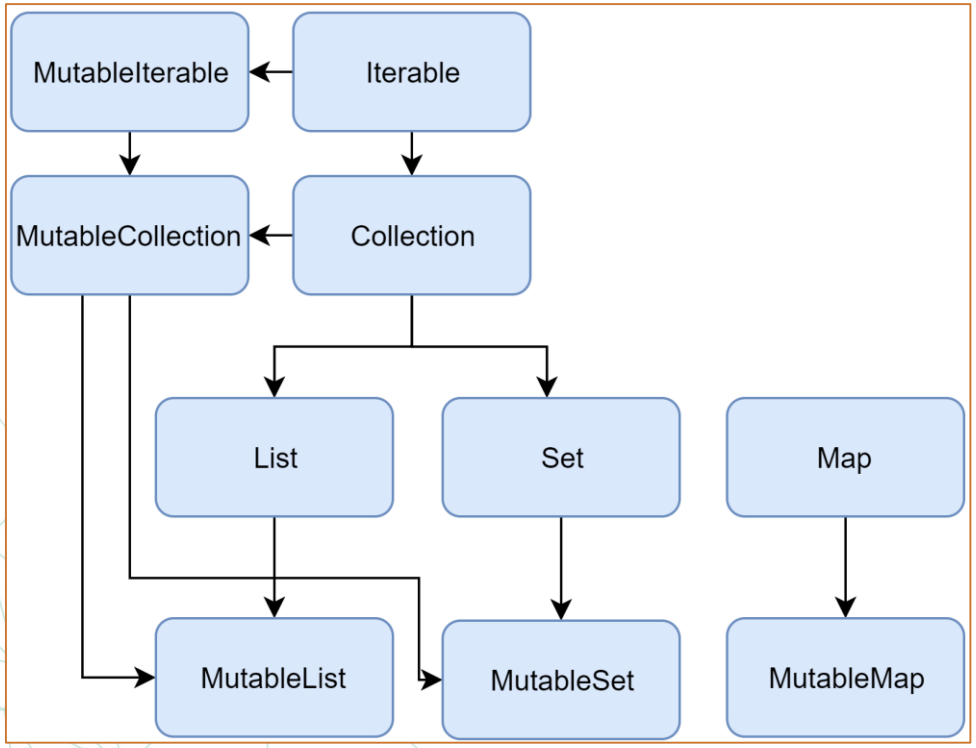
Abstract Class



Interface

Collections

- A collection usually contains a number of objects of the same type
- By using collections, you can build stronger systems with less code



https://kotlinlang.org/docs/collections-overview.html

List

- A list is an ordered, resizable collection like a dynamic (resizable) array
- ArrayList is cheap to read and generally cheap to add, but it is expensive to inject (insert elements in the middle) or remove.

```
fun main(args: Array<String>) {
    val numbers = mutableListOf(10)
    println(numbers.size)
    numbers.add(50)
    println(numbers.size)
    numbers.add(90)
    println(numbers.size)
    numbers.removeAt( index: 0)
    numbers.removeAt( index: 1)
    numbers.removeAt( index: 0)
    //Removes all elements from this collection
    numbers.clear()
    println(numbers.size)
}
```

https://kotlinlang.org/docs/list-operations.html

Visibility Modifiers

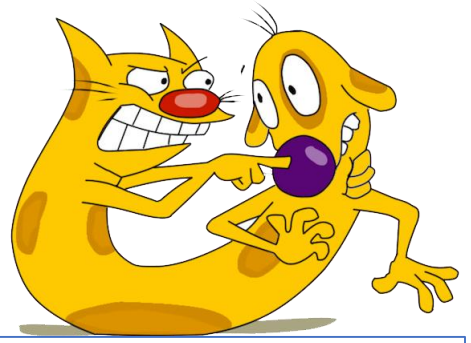
- public: Visible to the whole world (= Java)
- private: Visible to its own class (= Java)
- protected: Accessible inside the **class** and its **inheritors**
- internal: Accessible in the **module** (set of source files compiled together)

Task

Bsbs is having an argument with Boby about their favorite kind of food, help them by creating two classes Cat and the other one is Dog with the method eat() as a common method among them. Make an instance from each class then print the result from the above method.

```
fun main(args: Array<String>) {
    val c = Cat()
    c.eat()

    val d = Dog()
    d.eat()
}
```

output



```
Bsbs wants Fish
Boby wants Meat
```

SOLID Principles (Self Study)

- Single Responsibility Principle
- Open/Closed Principle
- Liskovs Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

A class should have one and only one reason to change

```
class UserWithValidation(val name: String, val email: String) {
    fun validateEmail(email: String) {
        // Email validation logic here
    }
}
```

```
// Improved class following SRP
class User(val name: String, val email: String)

class UserValidator {
    fun validateEmail(email: String) {
        // Email validation logic here
    }
}
```

Open/Closed Principle

Software entities should be open for extension, but closed for modification

```
enum class PaymentType { CASH, CREDIT_CARD }

class PaymentManager {
    fun pay(amount: Double, type: PaymentType) {
        if (type == PaymentType.CASH)
            println("Additional fees will be added")
        else if (type == PaymentType.CREDIT_CARD)
            println("Enter Card details")
    }

}
```

Open/Closed Principle

What if the client wants to add another payment option like “IN_STORE”

```
enum class PaymentType { CASH, CREDIT_CARD, IN_STORE }

class PaymentManager {
    fun pay(amount: Double, type: PaymentType) {
        when (type) {
            PaymentType.CASH ->
                println("Additional fees will be added")
            PaymentType.CREDIT_CARD ->
                println("Enter Card details")
            PaymentType.IN_STORE ->
                println("Applying 20% discount = ${amount * 0.2}")
        }
    }
}
```

Open/Closed Principle

```
interface Payable {
    fun pay(amount: Double)
}
```

```
class CreditCardPayment : Payable {
    override fun pay(amount: Double) {
        println("Enter Card details")
    }
}
```

```
class CashPayment : Payable {
    override fun pay(amount: Double) {
        println("Additional fees will be added")
    }
}
```

```
class InStorePayment : Payable {
    override fun pay(amount: Double) {
        println("Applying 20% discount = ${amount * 0.2}")
    }
}
```

Liskov Substitution Principle

A subclass should behave in such a way that it will not cause problems when used instead of the superclass

```
open class Bird {
    open fun fly() {}
}

class Penguin : Bird() {
    override fun fly() {
        print("Penguins can't fly!")
    }
}
```

Liskov Substitution Principle

```
// general bird methods and properties
open class Bird {}

//Unique Properties are separated
interface Flyable {
    fun fly()
}

// Penguins methods and properties
class Penguin : Bird() {}

class Eagle : Bird(), Flyable {
    override fun fly() {}
}
```

Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they don't use

```
interface NotificationService {
    fun sendEmailNotification(email: String, data: String)
    fun sendSMSNotification(phone: String, msg: String)
    fun sendPushNotification(uID: String, content: String)
}
```

```
class SMSOrderConfirmationService : NotificationService {
    override fun sendEmailNotification(email: String, data: String) {}
    override fun sendSMSNotification(phone: String, msg: String) {
        println("$phone: $msg")
    }
    override fun sendPushNotification(uID: String, content: String) {}
}
```

Interface Segregation Principle

```
interface EmailNotificationService {
    fun sendEmailNotification(email: String, data: String)
}
```

```
interface SMSNotificationService {
    fun sendSMSNotification(phone: String, msg: String)
}
```

```
interface PushNotificationService {
    fun sendPushNotification(uID: String, content: String)
}
```

```
class LowSignalConfirmationService :
    PushNotificationService, EmailNotificationService {
    override fun sendEmailNotification(email: String, data: String) {}
    override fun sendPushNotification(uID: String, content: String) {}
}
```

Dependency Inversion Principle

- High-level modules or classes should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions

Without DIP (Tight Coupling)

```
class UserManager {
    // Directly depends on DatabaseHelper
    private val helper = DatabaseHelper()
    fun registerUser(username: String, password: String) {
        helper.save( data: "$username: $password")
    }
}
```

```
class FileHelper {
    fun save(data: String) {
        // Save data to file
    }
}
```

```
class DatabaseHelper {
    fun save(data: String) {
        // insert data to DB
    }
}
```

With DIP (Loose Coupling)

```
interface DataStore {
    fun save(data: String)
}
```

```
class FileHelper : DataStore {
    override fun save(data: String) {
        // Save data to file
    }
}
```

```
class DatabaseHelper : DataStore {
    override fun save(data: String) {
        // insert data to DB
    }
}
```

```
class UserManager(private val helper: DataStore) {
    // Directly depends on DatabaseHelper
    fun registerUser(username: String, password: String) {
        helper.save( data: "$username: $password")
    }
}
```

```
fun main(args: Array<String>) {
    val file = UserManager(FileHelper())
    file.registerUser( username: "",  password: "")
    val db = UserManager(DatabaseHelper())
    db.registerUser( username: "",  password: "")
}
```

رواد مصر الرقمية

وزارة الاتصـــــالات
وتكنولوجيا المعلومات

Download Android Studio

Don’t forget to bring your USB cable

https://youtu.be/P8_y57etCOI

QUESTIONS?

THANK YOU

More Questions: WhatsApp Group