Discover the ease of Android development with Kotlin basics, a key component of our **Day 4 Android 14 Masterclass**. In this article, we cover the essentials, from the **utility of multiline comments** to the intricacies of **mutable and immutable lists**. Dive into the **distinctions between functions and methods** and how to **manipulate lists through indexing, adding, or removing elements**. Plus, we'll break down the syntax and practical applications of **loops** and explore the privacy of variables with '**private val**'.

1. Multiline Comments: Kotlin Basics

Multiline comments allow developers to leave **notes** or temporarily **disable blocks of code**. Enclosed between `/*` and `*/`, these comments can span multiple lines.

Syntax:

```
/*
This is a multiline comment
that spans multiple lines.
*/

```

2. ListOf and MutableListOf

`ListOf` creates an **immutable list**, whereas `MutableListOf` creates a **mutable list**, allowing modification like adding or removing items.

Syntax:

```
val immutableList = listOf("item1", "item2")
val mutableList = mutableListOf("item1", "item2")

```

3. Function vs. Method

Key Differences

- **Association**: Functions are stand-alone, and they are not associated with any object. Methods are associated with objects and are defined inside classes.
- **Calling**: Functions can be called directly by their name. Methods need to be called using the object they belong to.

Examples:

One simple predefined function in Kotlin is `println()`, used to print output to the standard output stream (usually the console).

```
fun main() {
    println("Hello, Kotlin!")
}

// Output: Hello, Kotlin!

```

String Method: `.toUpperCase()`

A common string method, which converts all the characters in a string to uppercase.

```
fun main() {
    val text = "hello, kotlin!"
    val uppercasedText = text.toUpperCase()
println(uppercasedText)
}

// Output: HELLO, KOTLIN!

```

In this example, the `toUpperCase()` method is called on the string `text`, converting all its characters to uppercase, and the result is then printed to the console.

4. Understanding Indexing in Lists

When you are working with collections in Kotlin, such as lists, you'll encounter the concept of "**indexing**." Indexing **refers to the position of elements** within the collection.

What is an Index?

An index is a numerical value that indicates the position of an element within a collection. Indexing **allows you** to a**ccess, modify, or remove elements** efficiently within collections like lists or arrays.

Key Points about Indexing:

- Indexing starts from `0`. The first element is at index `0`, the second element is at index `1`, and so on.
- The last index of a collection is `size of the collection - 1`.

Accessing Elements

You can access an element in a list by referring to its index.

Syntax:

```kotlin
val element = myList[index]

```

Example:

```kotlin
val myList = listOf("apple", "banana", "cherry")
println(myList[1]) // Output: banana

```

This will give you "banana" because index 1 corresponds to the second item.

Adding Items to a List: Kotlin Basics

- Since the regular lists (**`List`**) are immutable (can't be changed), you would use a mutable list (**`MutableList`**) to modify the content.
- Here is how you can add an item:

```
val mutableList = mutableListOf("apple", "banana", "cherry")
mutableList.add("date") // This adds "date" to the end of the list.

```

Removing Items from a List: Kotlin Basics

You can remove an item either by specifying the object to be removed or by its index.

```
mutableList.remove("banana") // This removes the item "banana" from the list.
mutableList.removeAt(0)
// This removes the first item ("apple") because it is at index 0.

```

Replacing Items in a List: Kotlin Basics

You can replace an item by assigning a new value to a specific index.

```
mutableList[1] = "blueberry"

// This replaces the item at index 1 ("cherry") with "blueberry".

```

5. Using the `set` Method: Kotlin Basics

The **`set`** method in Kotlin is used with mutable lists to replace an item at a specific position or index. Here's how you can use the **`set`** method to replace items in a list:

Syntax:

The **`set`** method requires two parameters: the index at which the item needs to be

The `set` method requires two parameters: the index at which the item needs to be replaced and the new value.

```
mutableList.set(index, element)

```

Example:

- Let’s say you have a mutable list of fruits, and you want to replace “banana” with “blueberry”.

```
val mutableList = mutableListOf("apple", "banana", "cherry")
mutableList.set(1, "blueberry")

// Replace the element at index 1 ("banana") with "blueberry".

```

Result:

- After executing the above code, the list **`mutableList`** would look like this:

```
["apple", "blueberry", "cherry"]

```

6. Using the `removeLast()` Method: Kotlin Basics

The **`removeLast()`** method is used in Kotlin to remove the last element from a mutable collection such as a **`MutableList`**.

This method modifies the original collection by removing the last element and returns the element that was removed.

If the collection is empty, calling this method will result in a **`NoSuchElementException`**.

Syntax:

```
mutableCollection.removeLast()

```

Example:

- Let’s say you have a mutable list of numbers, and you want to remove the last number.

```
val numbers = mutableListOf(1, 2, 3, 4, 5)
val removedNumber = numbers.removeLast()

// Removes 5 from the list and stores it in removedNumber.

```

Result:

- After executing the above code, the list **`numbers`** would look like this:

```
[1, 2, 3, 4]

```

And **`removedNumber`** would hold the value **`5`**.

7. Using the `contains()` Method: Kotlin Basics

The **`contains()`** method in Kotlin is used to check if a specific element is present within a collection, such as a list, set, or map. This method returns a boolean value: **`true`** if the element is found, and **`false`** if the element is not present in the collection.

Syntax:

```
collection.contains(element)

```

Example:

- Let’s say you have a list of fruits, and you want to check if “banana” is in the list.

```kotlin
val fruits = listOf("apple", "banana", "cherry")
val isBananaPresent = fruits.contains("banana") // Checks if "banana" is in the list

```

Result:

- `isBananaPresent` would hold the value `true` because "banana" is indeed in the list.

8. What is a For Loop?

A for loop is like a super-powered repeat button. It does something over and over again, like repeating a song, but with a twist. You can use it to do something slightly different each time, like singing the song in a different pitch.

Basic For Loop:

A simple for loop goes through a list of things one at a time and does something with each thing.

Example:

```kotlin
for (fruit in listOf("apple", "banana", "cherry")) {
    println(fruit)
}

```

This loop says: "For each fruit in this fruit basket, shout out (print) the name of the fruit."

Using Index in For Loop:

You can also loop through things by their position number (index) and get both the position number and the thing at that position.

- **Example:**

```kotlin
val fruits = listOf("apple", "banana", "cherry")

for ((index, fruit) in fruits.withIndex()) {
    println("Fruit at position $index is $fruit")
}

```

This loop says: “For each fruit in this fruit basket, shout out the position number and the name of the fruit.”

Using Break in For Loop:

The `break` statement is like a **stop button**. It stops the repeating action immediately, even if there were more things left to do.

- **Example:**

```kotlin
for (number in 1..10) {
    println(number)
    if (number == 5) {
       break
}
}

```

This loop says: “Count numbers from 1 to 10, but if you reach 5, stop counting.”

9. Understating private val: Kotlin Basics

In Kotlin, `private val` is used to define a **read-only property** that is accessible only within the class or file where it is declared.

Let’s break down what `private val` means:

`val`:

- **Immutable (Read-Only):** `val` means that the property is read-only, which means once it is assigned a value, it cannot be changed. It's like buying a toy that comes in a sealed box that you can look at but can't open or modify.

`private`:

- **Limited Access:** `private` is an access modifier that restricts the visibility of the property. If a property is marked as `private`, it can only be accessed within the class or file where it is declared. It's like having a private diary that only you can read.

Combining `private` and `val`:

- When you combine `private` and `val`, you get a property that can't be modified after it's set, and it's hidden from other classes or files. It's like having a personal, unchangeable secret.

Example:

```kotlin
class SecretKeeper {
      private val secretMessage: String = "The cake is a lie!"

      fun shareSecret(): String {
    // Accessible here because it's within the class
          return "I can tell you that: $secretMessage"
      }
}

// Outside the class, secretMessage is not accessible
// val message = SecretKeeper().secretMessage // This would cause an error

```

In this example, `secretMessage` is a private, read-only property of the `SecretKeeper` class. It can be used within the class but not modified, and it's hidden from the outside world.

Why Use `private val`?

- **Encapsulation:** It helps in keeping the internal state of a class hidden and unmodifiable from outside interference, promoting encapsulation and cleaner code.
- **Safety:** It prevents unintentional modifications and access, making the code safer and more predictable.

Conclusion. Kotlin Basics: For Loops and List Manipulations – Day 4 Android 14 Masterclass

Wrapping up Day 4 of our Android 14 Masterclass, we've explored the crucial Kotlin basics that underpin effective Android development. From understanding the ins and outs of list operations to implementing private values for secure coding, Kotlin offers a robust platform for building Android apps with finesse. As we've seen, Kotlin's intuitive syntax and versatile functions simplify the coding process, making it a preferred choice for modern developers. Keep these lessons as a beacon as you continue on your path to becoming an Android development expert, and harness the full potential of Kotlin in your future projects.

If you want to skyrocket your Android career, check out our **The Complete Android 14 & Kotlin Development Masterclass.** Learn Android 14 App Development From Beginner to Advanced Developer.

Master Jetpack Compose to harness the cutting-edge of Android development, **gain proficiency in XML** — an essential skill for numerous development roles, and **acquire practical expertise by constructing real-world applications**.

Check out Day 2 of this course here.