


Exploring Basic Kotlin Syntax and Structure - Day 2 Android 14 Masterclass

 Jafar Jabbarzadeh / 9. November 2023 / Android



Exploring Basic Kotlin Syntax and Structure - Day 2 Android 14 Masterclass

Welcome to Day 2 of our Android 14 Masterclass, where we dive into the **Basic Kotlin Syntax and Structure**. As you embark on this coding journey, understanding Kotlin's syntax and structure is paramount. Today, we're peeling back the layers of this intuitive language, starting with variables and data types, maneuvering through user input and control flows, and mastering loops and operators. Whether you're a seasoned developer or a novice in the app-making realm, our guide will solidify your foundation and equip you with the essential tools for crafting robust Android applications.

1. What are Variables?

Variables are fundamental building blocks in Kotlin programming (and programming in general), allowing developers to store, modify, and manage data within their applications. Variables can hold various types of data, such as numbers, characters, strings, or even more complex data structures like lists or objects.

Creating Variables: Exploring Basic Kotlin Syntax and Structure

In Kotlin, variables can be defined using either **var** or **val** keyword, followed by the **variable name**, an optional **data type**, and an **assignment**. The **assignment operator** is represented by the equals sign (**=**). The assignment operator is used to assign a value to a variable.

```
var variableName: DataType = value // Mutable variable (can change value)
val constantName: DataType = value // Immutable (Read-only) variable
```

More examples:

```
var age: Int = 30 // Mutable integer variable
val pi: Double = 3.14 // Immutable double variable
var name = "John Doe" // Type inferred as String, Mutable
val isAdult = true // Type inferred as Boolean, Immutable
```

val vs. **var** :

val: Stands for “**value**” and it’s **immutable**, which means once you assign a value to a **val** variable, you cannot change or reassign it.

Preferred when you have a variable whose value shouldn’t change once initialized, like **constants** or **properties that should remain unchanged**.

```
val pi = 3.14 // An immutable variable
// pi = 3.14159 // This would cause a compilation error
```

var: Is mutable, meaning after you assign an initial value, you can change or reassign that variable to a new value as many times as you want.

Used when you anticipate the value of **a variable will change**, like counters in a loop or a **value being updated** based on user input.

```
var counter = 0 // A mutable variable
counter = 1 // Modifying the value of the variable
```

2. What are Datatypes?

When you're programming, you work with **different kinds of information or data**, like numbers, words, or true/false values. Datatypes in Kotlin are like labels that tell the computer what kind of data you're dealing with so it knows how to handle them properly.

You need to tell the computer the datatype of your information (like number, word, true/false) when you first create a variable. Once you set the type, it stays the same.

Integers (**Int** and **Long**)

Description: Integer types can hold **whole numbers**, both positive and negative. The most commonly used integer type is **Int**. For larger integer values, **Long** can be used.

Syntax and Examples:

```
val age: Int = 25
```

```
val largeNumber: Long = 100000000000L
```

Floats and Doubles (Float and Double)

Description: Floats and Doubles are used to represent **decimal numbers**. **Double** has higher precision and is generally used as the **default** for decimal numbers.

Syntax and Examples:

```
val pi: Double = 3.14  
val floatNumber: Float = 2.73F
```

Booleans (Boolean)

Description: Booleans are like light switches in programming. They **can only have two values**: **true** (on) or **false** (off). They are used to make decisions in code, allowing parts of your program to run based on whether a condition is true or false.

Logical Operators:

- **||** (Logical OR): Returns true if at least one condition is true.
- **&&** (Logical AND): Returns true only if both conditions are true.
- **!** (Logical NOT): Negates the value; turns true into false and vice versa.

Syntax and Examples:

```
val isTrue = true  
val isFalse = false
```

```
val result1 = isTrue || isFalse // Logical OR, result1 will be true
val result2 = isTrue && isFalse // Logical AND, result2 will be false
val result3 = !isTrue // Logical NOT, result3 will be false
```

| vs. || Operators

Both `|` and `||` operators are used with boolean values (true or false) in Kotlin, but **they serve different purposes and behave differently**.

- **Evaluation:**

- `|` always evaluates both operands.
- `||` performs short-circuit evaluation, skipping the second operand if the first is `true`.

- **Use Cases:**

- `|` is versatile, used for bitwise or logical OR operations.
- `||` is solely used for logical OR operations to control the flow of programs based on conditions.

- **Efficiency:**

- `||` can be more efficient due to short-circuit evaluation, as it may skip the evaluation of the second operand.
- `|` might be slightly less efficient in logical operations as it always evaluates both operands.

Char

The `char` data type in Kotlin represents a single character. A character could be a letter, a number, a punctuation mark, or even a symbol like \$ or &. Let's break it down!

Characters are surrounded by single quotes (`' '`).

```
val letter: Char = 'A'  
val number: Char = '1'  
val symbol: Char = '$'
```

Unicode Characters:

Characters in Kotlin can also represent Unicode characters, allowing you to use a wide range of symbols and letters from various languages.

Example:

```
val heart: Char = '\\u2764'  
println(heart) // Output: ♥
```

Special Characters:

There are also special escape characters in Kotlin, which allow you to represent special values such as a new line or a tab.

Example:

- New line: '\\n' - moves to the next line
- Tab: '\\t' - adds a tab space

Strings

Strings are **sequences of characters** enclosed within double quotes (" "). They are used to manage and **manipulate text** in Kotlin.

Strings are immutable, meaning once a string is created, it cannot be changed, but you can create a new string based on modifications of the original string.

```
val simpleString: String = "Hello, World!"
```

String Concatenation:

Strings can be joined together using the `+` operator.

```
val firstName = "John"
val lastName = "Doe"
val fullName = firstName + " " + lastName // "John Doe"
```

3. Type Conversion

Type conversion, also known as **type casting**, is a process where the **type of a variable is converted to another datatype**. Explicit conversion is often necessary when you want to work with variables of incompatible types together.

Common Type Conversions

Numbers:

You might want to convert between different number types, like from `Int` to `Double`, or `Long` to `Int`.

```
val integer: Int = 5
val double: Double = integer.toDouble() // converting Int to Double
```

Numbers to Strings:

Numbers can be converted to strings when you want to display them as text or concatenate them with other strings.

```
val number: Int = 42
val stringNumber: String = number.toString() // converting Int to String
```

Strings to Numbers:

If a string contains a number, you can convert that string into an actual number type, like `Int` or `Double`.

```
val stringNumber = "123"
val intNumber: Int = stringNumber.toInt() // converting String to Int
```

4. User Input

Getting input from the user is essential in programming **when you want your application to interact with the user** by receiving data that the user provides. In Kotlin, you can receive user input from the console using the standard library functions, making your programs interactive and dynamic.

If you want to ask the user for input and display it, you can use the combination of `println()` to show a message and `readLine()` to get the user's response.

Here's a simple way to do it:

```
println("What is your name?") // Display a message to the user with println()

val userName = readLine() // Get the user's response with readLine()
```



```
println("Hello, $userName!") // Show the user's response with println()
```

5. Basic Kotlin Syntax - Understanding `if` and `else if` Statements

`if` and `else if` statements are used in Kotlin to make decisions in your code based on **conditions**. They allow your program to execute different blocks of code based on whether certain conditions are true or false.

- **`if` Statement:**

An `if` statement checks a condition and executes a block of code **if that condition is true**.

```
val age = 20

if (age >= 18) {
    println("You are eligible to vote.")
}
```

In this example, since the age is 20 (which is greater than or equal to 18), the message "You are eligible to vote." will be printed to the console.

- **The `if-else` Statement:**

You can use an `else` statement following an `if` statement to execute a block of code when the condition in the `if` statement is false.

```
val age = 15

if (age >= 18) {
```

```
println("You are eligible to vote.")
} else {
println("You are not eligible to vote.")
}
```

Here, because the age is 15 (which is less than 18), the condition in the **if** statement is false, so the message "You are not eligible to vote." will be printed.

- **The else if Statement:**

An **else if** statement follows an **if** statement and checks another condition **if the previous condition was false**.

```
val number = 0

if (number > 0) {
println("The number is positive.")
} else if (number == 0) {
println("The number is zero.")
}
```

In this example, since the number is not greater than 0, it checks the next condition (**number == 0**), which is true, so it prints "The number is zero."

- **Combining if, else if, and else:**

You can combine **if**, **else if**, and **else** to handle multiple conditions and a default case.

```
val number = -5

if (number > 0) {
println("The number is positive.")
} else if (number < 0) {
println("The number is negative.")
}
```

```
} else {  
    println("The number is zero.")  
}
```

Here, the program checks multiple conditions:

- If the number is positive, it prints "The number is positive."
- If the number is negative (which it is), it prints "The number is negative."
- If none of the above conditions are true, it defaults to printing "The number is zero."

6. Basic Kotlin Syntax - Understanding **when** Statements

- **when** is like a decision-maker in your code.
- You give **when** a value, and it picks a result based on that value.
- It makes your code cleaner and easier to understand when you have many choices.

So, you can think of the **when** statement as a smarter, more organized way of making decisions in your code compared to the **if** statement!

Example:

```
val number = 2  
  
when (number) {  
    1 -> println("It is one.")  
    2 -> println("It is two.")  
    3 -> println("It is three.")  
    else -> println("It is not one, two, or three.")  
}
```

7. Basic Kotlin Syntax - Understanding `while` Loops

Imagine you have a robot that can repeat a task for you, like clapping hands or counting numbers. A `while` loop in Kotlin is like that robot. It keeps doing a task over and over as long as a certain condition is true.

How It Works:

Let's say you want the robot to clap hands five times. You can tell the robot:

1. Start counting from one.
2. Clap your hands.
3. If you haven't clapped five times, go back to step 2.

In Kotlin, you write this like:

```
var count = 1 // Step 1: Start counting from one
```

```
while (count <= 5) { // As long as you haven't clapped five times
    println("Clap hands!") // Step 2: Clap your hands
    count++ // Go back to step 2 if you haven't clapped five times
}
```

Warning: Be Careful of Infinite Loops!

If you forget to tell the robot **when to stop** (like forgetting the `count++`), the robot will keep clapping hands **forever**!

This is called an “infinite loop,” and it can make your program run forever and not work correctly.

So, a **while** loop helps you repeat tasks in your code, but remember to always give it a stopping condition! More examples:

Basic Kotlin Syntax for Counting to Ten

```
var number = 1
while (number <= 10) {
    println("Number: $number")
    number++ // Don't forget this part! It makes sure the loop won't go on forever.
}
```

In this example, as long as the variable **number** is less than or equal to ten, it will keep printing the number and then adding one to it.

Countdown

```
var countdown = 5
while (countdown > 0) {
    println("Countdown: $countdown")
    countdown--
}
println("Blast off!")
```

Here, the loop starts at five and counts down, subtracting one each time until it gets to one. Then it prints "Blast off!"

8. Basic Kotlin Syntax - Operators

Operators are **special symbols** or **keywords** that are used to perform operations. You have already seen some operators in the boolean section of this summary and in the code examples.

Let's discuss some common categories of operators:

Arithmetic Operators

These are used for mathematical operations:

- **Addition** `+`: `5 + 3` results in `8`
- **Subtraction** `-`: `5 - 3` results in `2`
- **Multiplication** `*`: `5 * 3` results in `15`
- **Division** `/`: `5 / 2` results in `2.5`
- **Modulus** `%`: `5 % 2` results in `1` (Remainder of 5 divided by 2)

Comparison Operators

These operators compare two values:

- **Equals** `==`: `5 == 3` results in `false`
- **Not Equals** `!=`: `5 != 3` results in `true`
- **Greater Than** `>`: `5 > 3` results in `true`
- **Less Than** `<`: `5 < 3` results in `false`
- **Greater Than or Equal To** `>=`: `5 >= 5` results in `true`
- **Less Than or Equal To** `<=`: `5 <= 3` results in `false`

Assignment Operators

Used to assign values to variables:

- **Equals** `=`: `val x = 5`
- **Plus Equals** `+=`: `x += 3` (Equivalent to `x = x + 3`)
- **Minus Equals** `-=`: `x -= 3` (Equivalent to `x = x - 3`)
- **Multiply Equals** `*=`: `x *= 3` (Equivalent to `x = x * 3`)
- **Divide Equals** `/=`: `x /= 3` (Equivalent to `x = x / 3`)

Logical Operators

Used for boolean logic (true or false values):

- AND `&&`: `true && false` results in `false`
- OR `||`: `true || false` results in `true`
- NOT `!`: `!true` results in `false`

Conclusion: Exploring Basic Kotlin Syntax and Structure - Day 2 Android 14 Masterclass

As we wrap up **Day 2 of our Android 14 Masterclass**, we've journeyed through the essential Basic Kotlin Syntax and structures that make Kotlin such a powerful and efficient tool for developers. From the versatility of variables to the logic of control statements and the persistence of loops, we've covered a breadth of fundamental concepts that are crucial for creating dynamic Android apps. Remember, these building blocks are just the starting point; with practice, you'll continue to hone your skills and unlock the full potential of Kotlin in your Android development adventures. Keep experimenting, coding, and learning - your path to becoming a Kotlin connoisseur is well underway!

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. Build Apps like Trello, 7Min Workout, Weather App.

Check out Day 3 of this course [here](#).

Related Posts:

1. [Basic Kotlin Syntax: Functions, Objects and Classes - Day 3 Android 14 Masterclass](#)
2. [Mastering Screen Navigation with Kotlin - Day 10 Android 14 Masterclass](#)
3. [Integrating Location Services in Your Android App - Day 11 Android 14 Masterclass](#)
4. [Mastering UI Design with Jetpack Compose - Days 15-16 Android 14 Masterclass](#)

Tags:

[ANDROID 14](#)

[BOOLEANS](#)

[CHAR](#)

[DATATYPES](#)

[FLOATS AND DOUBLES](#)

[IF AND ELSE IF](#)

[INT AND LONG](#)

[KOTLIN](#)

[KOTLIN SYNTAX](#)

[OPERATORS](#)

[TYPE CONVERSION](#)

[USER INPUT](#)

[VAL VS. VAR](#)

[VARIABLES](#)

[WHEN STATEMENTS](#)

[WHILE LOOPS](#)

[Collaboration](#)

[Media Kit](#)

[Contact](#)

[Imprint](#)

[About Us](#)

[Privacy Policy](#)

©2024 TutorialsEU

Neve | Powered by WordPress