Since you're following CheezyCode, here are the unique videos from Telusko that are not covered in CheezyCode, so you can avoid repeated content:

1. **String Comparison** – Explains how to compare strings in Kotlin.
   * [#12 Kotlin Tutorial | String Comparison (4:29)](https://youtu.be/STRING-COMPARISON)
2. **Decompile Bytecode** – Introduces decompiling Kotlin code into Java bytecode.
   * [#8 Kotlin Tutorial | Decompile Bytecode (6:02)](https://youtu.be/DECOMPILE-BYTECODE)
3. **Kotlin and Java Together** – Shows how Kotlin interacts with Java.
   * [#6 Kotlin Tutorial | Kotlin Java Together (5:28)](https://youtu.be/KOTLIN-JAVA)
4. **Backtick as Escape Character** – Special topic about the use of backticks in Kotlin.
   * [#38 Kotlin Tutorial | Backtick As Escape Character (3:39)](https://youtu.be/BACKTICK-ESCAPE)
5. **Anonymous Inner Class** – Explanation on creating anonymous inner classes.
   * [#35 Kotlin Tutorial | Anonymous Inner Class (6:39)](https://youtu.be/ANONYMOUS-INNER-CLASS)
6. **Recursion** – Detailed coverage of recursion and tail recursion with factorial examples.
   * [#24 Recursion (7:53)](https://youtu.be/RECURSION)
   * [#25 Tail Recursion (6:57)](https://youtu.be/TAIL-RECURSION)
7. **String to Integer Conversion** – Standalone explanation on how to convert data types.
   * [#20 String to Integer (4:44)](https://youtu.be/STRING-INTEGER)
8. **User Input** – Covers how to handle user input in Kotlin.
   * [#39 Kotlin Tutorial | User Input in Kotlin (5:12)](https://youtu.be/USER-INPUT)

**Kotlin – Basics**

**1. Variables and Values**

* **Variables (var)**: Mutable references, their values can be changed.

var name = "John"

name = "Doe"

* **Values (val)**: Immutable references, their values cannot be reassigned after initialization.

val age = 25

// age = 30 // Error: Val cannot be reassigned

**2. Types**

* Kotlin is a statically typed language; the type of variable is determined at compile time.
* Common types include:
  + Int, Double, Float, Long, Short, Byte for numbers
  + Char for characters
  + String for text
  + Boolean for logical values (true or false)

**3. Numbers**

* **Integer Types**: Byte (8 bits), Short (16 bits), Int (32 bits), Long (64 bits)

val num: Int = 10

val bigNum: Long = 100000L

* **Floating-Point Types**: Float (32 bits), Double (64 bits)

val pi: Float = 3.14F

val e: Double = 2.71828

**4. Char and String**

* **Char**: Represents a single character; declared using single quotes.

val letter: Char = 'A'

* **String**: Represents a sequence of characters; declared using double quotes.
  + Supports string interpolation with $ sign.

val greeting = "Hello"

val name = "Alice"

val message = "$greeting, $name!" // "Hello, Alice!"

**5. Boolean**

* Represents logical values: true or false.

val isKotlinFun: Boolean = true

**6. Logical Operations**

* **AND (&&)**: Returns true if both operands are true.

val result = true && false // false

* **OR (||)**: Returns true if at least one operand is true.

val result = true || false // true

* **NOT (!)**: Returns the opposite of the operand's value.

val result = !true // false

**Short Circuiting in Logical Operators (Kotlin)**

* **Logical AND (&&)**: If the first condition is false, the second condition is **not evaluated**.
  + Example: if (a > 5 && b < 10)
    - If a > 5 is false, b < 10 won't be checked.
* **Logical OR (||)**: If the first condition is true, the second condition is **not evaluated**.
  + Example: if (a == 5 || b > 10)
    - If a == 5 is true, b > 10 won't be checked.

Conditional Statements :

**if Statement**

* Used to execute a block of code if a condition is true.

val number = 10

if (number > 0) {

println("Positive number")

}

**2. if-else Statement**

* Provides an alternative block of code to execute if the condition is false.

val number = -5

if (number > 0) {

println("Positive number")

} else {

println("Negative number")

}

**3. if-else if Ladder**

* Allows multiple conditions to be checked in sequence.

val number = 0

if (number > 0) {

println("Positive number")

} else if (number < 0) {

println("Negative number")

} else {

println("Zero")

}

* **if Expression**: if can return a value and be used as an expression.

val max = if (a > b) a else b

**Ranges in Kotlin**

* **Range syntax**: a..b (includes both a and b).
  + Example: 1..5 → 1, 2, 3, 4, 5.
* **DownTo**: a downTo b (decreasing order).
  + Example: 5 downTo 1 → 5, 4, 3, 2, 1.
* **Step**: Controls the increment.
  + Example: 1..5 step 2 → 1, 3, 5.

**when in Kotlin**

* **Syntax**: Used as a replacement for switch cases.
  + Example:

when (x) {

1 -> print("One")

2 -> print("Two")

else -> print("Other")

}

**when as an Expression**

* Can return a value.
  + Example:

val result = when (x) {

1 -> "One"

2 -> "Two"

else -> "Other"

}

* **When without an argument**: Acts like an if-else chain.
  + Example:

when {

x % 2 == 0 -> print("Even")

else -> print("Odd")

}

LOOPS :

**1. while Loop**

* Repeatedly executes a block of code as long as the given condition is true.

var count = 1

while (count <= 5) {

println("Count: $count")

count++

}

// Output: Count: 1 to 5

**2. do-while Loop**

* Similar to the while loop, but guarantees at least one execution of the loop body, as the condition is checked after the block is executed.

var count = 1

do {

println("Count: $count")

count++

} while (count <= 5)

// Output: Count: 1 to 5

**3. for Loop**

* Used for iterating over a range, array, or any iterable object.
* **Iterating Over a Range**

for (i in 1..5) {

println("Number: $i")

}

// Output: Number: 1 to 5

* **Iterating with a Step**

for (i in 1..10 step 2) {

println(i)

}

// Output: 1, 3, 5, 7, 9

* **Iterating in Reverse**

for (i in 10 downTo 1) {

println(i)

}

// Output: 10 to 1

* **Iterating Over a Collection**

val items = listOf("Apple", "Banana", "Cherry")

for (item in items) {

println(item)

}

// Output: Apple, Banana, Cherry

**String Templating in Kotlin**

* **Embedding variables**: Use $variable to embed variables in strings.
  + Example: val name = "Kotlin"; println("Hello, $name!") → Outputs: Hello, Kotlin!
* **Expressions in templates**: Use ${expression} for more complex expressions.
  + Example: println("2 + 2 = ${2 + 2}") → Outputs: 2 + 2 = 4.

**4. Nested Loops**

* A loop inside another loop. The inner loop completes all its iterations for each iteration of the outer loop.

for (i in 1..3) {

for (j in 1..3) {

print("($i,$j) ")

}

println()

}

// Output:

// (1,1) (1,2) (1,3)

// (2,1) (2,2) (2,3)

// (3,1) (3,2) (3,3)

**5. break and continue**

* **break**: Exits the loop immediately.

for (i in 1..5) {

if (i == 3) break

println(i)

}

// Output: 1, 2

* **continue**: Skips the current iteration and continues with the next one.

for (i in 1..5) {

if (i == 3) continue

println(i)

}

// Output: 1, 2, 4, 5

**Functions in Kotlin**

* **Basic Syntax**:
  + Syntax: fun functionName(param1: Type, param2: Type): ReturnType { }
  + Example:

fun add(a: Int, b: Int): Int {

return a + b

}

* **Single-expression function**:
  + Syntax: fun functionName(param: Type) = expression
  + Example: fun square(x: Int) = x \* x

**Default Arguments**

* **Define default values**: Provides default values to function parameters.
  + Syntax: fun functionName(param1: Type = defaultValue) { }
  + Example:

fun greet(name: String = "Guest") {

println("Hello, $name!")

}

* + Calling greet() will print "Hello, Guest!", and greet("John") will print "Hello, John!"

### Function Overloading in Kotlin

* **Definition**: Multiple functions with the same name but different parameters.
* **Syntax**:

fun add(a: Int, b: Int): Int = a + b

fun add(a: Double, b: Double): Double = a + b

* Example:

println(add(2, 3)) // Calls Int version

println(add(2.5, 3.5)) // Calls Double version

**Storing Functions in Variables**

* **Definition**: Functions can be assigned to variables as objects.
* **Syntax**:

val functionName: (Type) -> ReturnType = ::functionName

* Example:

val sum: (Int, Int) -> Int = ::add

println(sum(5, 10)) // Output: 15

**Named Arguments**

* **Definition**: You can specify arguments by their parameter names to improve readability or change the order.
* **Syntax**:

fun displayDetails(name: String, age: Int) {

println("Name: $name, Age: $age")

}

displayDetails(age = 25, name = "John")

* Example:
  + You can call displayDetails(age = 25, name = "John"), making the code clearer.

**Variable Scope**

* **Local Scope**: Variables declared inside a function or a block are accessible only within that function or block.

fun example() {

val localVariable = 10

println(localVariable) // Accessible here

}

// println(localVariable) // Error: Unresolved reference

* **Global Scope**: Variables declared outside any function or block are accessible throughout the entire file.

val globalVariable = 100

fun showGlobal() {

println(globalVariable) // Accessible here

}

* **Variable Shadowing**: A local variable can have the same name as a global variable, but the local variable will "shadow" or override the global one within its scope.

val x = 5 // Global scope

fun shadowExample() {

val x = 10 // Local scope

println(x) // Prints 10

}

println(x) // Prints 5

**3. Local Functions**

* Functions defined inside another function. They are useful for creating helper functions that are only needed within the outer function.

fun outerFunction() {

fun localFunction(message: String) {

println(message)

}

localFunction("Hello from Local Function")

}

### Arrays in Kotlin

* **Definition**: A collection of elements of the same type.
* **Array Creation**:
  + Syntax: val arrayName = arrayOf(element1, element2, ...)
  + Example:

val numbers = arrayOf(1, 2, 3, 4)

* **Accessing Elements**:
  + Syntax: arrayName[index]
  + Example:

println(numbers[0]) // Outputs: 1

* **Modifying Elements**:
  + Syntax: arrayName[index] = value
  + Example:

numbers[1] = 10

* **Array with Specific Type**:
  + Syntax: val arrayName = arrayOf<Type>(element1, element2, ...)
  + Example:

val strings = arrayOf<String>("Kotlin", "Java")

* **Array Size**: Use array.size to get the number of elements.

**1. Classes and Objects**

* **Class**: A blueprint for creating objects, encapsulating data (properties) and functions (methods).
* **Object**: An instance of a class.
* **Defining a Class**:

class Person {

var name: String = ""

var age: Int = 0

}

* **Creating an Object**:

val person = Person() // Creating an object of the Person class

person.name = "John"

person.age = 30

**2. Properties and Primary Constructor**

* **Properties**: Variables defined inside a class. They can have default values or be initialized through constructors.
* **Primary Constructor**: A concise way to define properties and initialize them directly in the class header.
* **Example with Primary Constructor**:

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

val person = Person("Alice", 25)

println(person.name) // Alice

println(person.age) // 25

**3. Methods**

* **Methods**: Functions defined within a class that can operate on the class's properties.
* **Defining Methods**:

class Person(val name: String, var age: Int) {

fun greet() {

println("Hello, my name is $name.")

}

}

val person = Person("Alice", 25)

person.greet() // Outputs: Hello, my name is Alice.

Constructors

**lateinit in Kotlin**

* Used for **non-null** properties that will be initialized later.
* Can only be used with var (mutable) properties.
* Applicable only to properties declared **outside** constructors (i.e., class properties).
* Accessing an uninitialized lateinit variable throws an **UninitializedPropertyAccessException**.

lateinit var name: String

fun initializeName() {

name = "John"

}

**Getters and Setters in Kotlin**

* **By default**, Kotlin provides a getter and setter for every var property and only a getter for val.
* You can **override** them to provide custom logic.

**Default**

var name: String = "John"

// Implicit getter and setter

**Custom Getter and Setter**

var name: String = "John"

get() = field.uppercase() // Custom getter

set(value) {

field = value.trim() // Custom setter

}

* field is a **backing field** used to avoid recursion in getter/setter.

### Inheritance in Kotlin

* Use **open** to allow a class to be inherited.
* Child class can inherit properties and methods from **one parent** class.
* **Parent constructor** is called **first**, followed by the child constructor.

open class Parent {

fun myMethod() = println("I am in the main method")

}

class Child : Parent() {

fun myMethod2() = println("I am in child")

}

**Kotlin Overriding –**

1. **Inheritance Basics**:
   * open keyword: Needed to allow a class or method to be inherited or overridden.
   * Example: open class Mobile allows subclassing.
2. **Overriding Properties and Methods**:
   * Use override keyword in the subclass to redefine properties or methods of the superclass.
   * Example: override val name: String = "Oneplus Mobile"
3. **Superclass Reference**:
   * super keyword: Accesses superclass properties or methods within an overridden method.
   * Example: super.Display() calls Display from Mobile within OnePlus.
4. **toString() Override**:
   * Customize toString() for better object string representation.
   * Example: override fun toString(): String = "$name and $size"
5. **Constructor in Inheritance**:
   * Subclasses must pass arguments to superclass if it has a primary constructor.
   * Example: OnePlus(typeParam: String) : Mobile(typeParam)

**Example Recap:**

open class Mobile(val type: String) {

open val name: String = "Mobile"

open fun Display() = println("Mobile Display")

}

class OnePlus(typeParam: String) : Mobile(typeParam) {

override val name: String = "Oneplus Mobile"

override fun Display() {

super.Display() // Calls `Display` from `Mobile`

println("Oneplus display")

}

}

**Kotlin Polymorphism –**

1. **Polymorphism**: Enables classes to provide unique implementations of a method inherited from a common superclass.
2. **Parent Reference for Subclass Object**: A parent class variable can hold references to objects of any subclass.
3. **Method Overriding**: Subclasses provide specific implementations of a method from the superclass, which is determined at runtime.
4. **Loose Coupling**: Functions that accept the superclass type (e.g., Shape) can work with any subclass, increasing flexibility and extensibility.

**Code with Inline Comments**

fun main() {

// Parent class references (Shape) can hold child class objects (Circle, Square).

// This allows flexible use of different Shape types in one array.

val circle: Shape = Circle(4.0)

val square: Shape = Square(4.0)

// An array of Shape types - can hold any subclass of Shape

val shapes = arrayOf(Circle(3.0), Circle(4.0), Square(4.0), Triangle(3.0, 2.0))

calArea(shapes) // Loose coupling: calArea can work with any subclass of Shape

}

// Function to calculate areas of multiple Shape objects

fun calArea(shapes: Array<Shape>) {

// Dynamic dispatch: Each shape’s area() method is called based on its actual type

for (shape in shapes) {

println(shape.area())

}

}

// Base class Shape

open class Shape() {

open fun area(): Double = 0.0 // Default implementation

}

// Circle subclass overriding area() for its specific calculation

class Circle(val radius: Double) : Shape() {

override fun area(): Double = PI \* radius \* radius

}

// Square subclass overriding area() for its specific calculation

class Square(val side: Double) : Shape() {

override fun area(): Double = side \* side

}

// Triangle subclass overriding area() for its specific calculation

class Triangle(val base: Double, val height: Double) : Shape() {

override fun area(): Double = base \* height \* 0.5

}

**Kotlin Visibility Modifiers – Concept Notes**

1. **public** (default):
   * Accessible from anywhere in the project.
   * Classes, functions, properties, and constructors are public by default if no modifier is specified.
   * Example: public class MyClass (or just class MyClass).
2. **private**:
   * Limits accessibility to the same **class** (for classes, properties, functions).
   * If applied to a **top-level declaration** (class or function outside any class), it’s accessible only within the same **file**.
   * Example: private val x = 10 in a class restricts x to that class only.
3. **protected**:
   * Accessible within the class and its **subclasses**.
   * Only valid inside classes (not for top-level declarations).
   * More restrictive than public, but allows controlled access in inheritance.
   * Example: protected fun show() can be accessed by subclasses of the declaring class.
4. **internal**:
   * Limits visibility to the **same module** (e.g., a project or library module).
   * Useful for restricting visibility to parts of the codebase while allowing use across files within a module.
   * Example: internal class MyClass makes it accessible to the same module but hides it from other modules.

**Quick Summary Table**

| **Modifier** | **Visibility Scope** |
| --- | --- |
| **public** | Everywhere |
| **private** | Declaring class or file (top-level only) |
| **protected** | Declaring class and subclasses |
| **internal** | Within the same module |