Synopsis of Kotlin

Annotations: In Kotlin, annotations are a powerful mechanism for adding metadata to code elements, such as classes, functions, parameters, and properties, without affecting their actual functionality.

Inlining: By "inlining" function code, Kotlin improves performance by reducing the overhead associated with function calls. It's like having an efficient assistant who anticipates your needs, works alongside you, and streamlines the entire process.

Regex: A regular expression, often shortened to "regex" or "regexp," is a sequence of characters that defines a search pattern. Regular expressions are used to match patterns in strings and are a powerful tool for text processing. They can be used for tasks such as validation, search-and-replace, and text parsing.

Triple: As the code below shows, Triple is a data class. That is to say, Triple is final and cannot have a subclass. Furthermore, Triple is immutable since all three properties are defined as `val`. We can put three values of different types in a Triple, for example, `Triple(42, "Kotlin", Long.MAX_VALUE)`.

Destructuring: Kotlin's destructuring declarations allow you to extract fields from an object into a set of variables, which you can then use independently. Under the hood, the compiler uses `componentN` operator functions to unpack the values of an object and assign them to the variables.

Operator Overloading: Operator overloading in Kotlin allows you to provide custom implementations for a predefined set of operators on types. These operators have predefined symbolic representations (like + or `*`) and precedence.

Higher-Order Function: In the Use function types and lambda expressions in Kotlin codelab, you learned about higher-order functions, which are functions that take other functions as parameters and/or return a function, such as `repeat()`.

Implementation:

```
fun main() {
   val pattern = Regex("ll") // match ll
   val res: MatchResult? = pattern.find("Hello Hello", 5)
   println(res?.value)
}

fun main() {
   val obj = Triple(1, 2, 3)
   println(obj.toList())
}
```

```
data class Data(val name: String, val age: Int)

fun sendData(): Data {
    return Data("Michael", 50)
}

fun main() {
    val obj = sendData()
    println("Name is ${obj.name}")
    println("Age is ${obj.age}")

    // Destructuring objects
    val (name, age) = sendData()
    println("$name $age")
}

class Object(var objName: String) {
    // Overloading the function
    operator fun plus(b: Int) {
        objName = "Name is $objName and data is $b"
    }

    override fun toString(): String {
        return objName
    }
}

fun main() {
    val obj = Object("Michael")
    obj + 10
    println(obj)
}

fun hof(str: String, myCall: (String) -> Unit) {
    myCall(str)
}

fun main() {
    println("Result:")
    hof("Michael", ::println)
}
```