Fresher Android

Kotlin OOP Basic

Kotlin Object Oriented Programming Basic concept

#1. Kotlin OOP - Class

Declare a class

```
class NewClassName: ParentClass {
                    // Properties
                    // Methods
}
```

#1. Kotlin OOP - Class

Adding Properties to a Class

```
class BankAccount {
var accountBalance: Double = 0.0
var accountNumber: Int = 0
}
```

Defining Methods

```
class BankAccount {
        var accountBalance: Double = 0.0
        var accountNumber: Int = 0

        fun displayBalance()
        {
                println("Number $accountNumber")
                println("Current balance is $accountBalance")
        }
}
```

#1. Kotlin OOP - Class

Declaring and Initializing a Class Instance

```
val account1: BankAccount = BankAccount()
val account1 = BankAccount()
```

Primary and Secondary Constructors

```
class BankAccount {

        var accountBalance: Double = 0.0
        var accountNumber: Int = 0

        constructor(number: Int, balance: Double) {
        accountNumber = number
        accountBalance = balance
}
```

Initializer Blocks

```
class BankAccount (val accountNumber: Int, var accountBalance: Double) {

init {
// Initialization code goes here
}
```

#1. Kotlin OOP - Class

Calling Methods and Accessing Properties

```
classInstance.propertyname
classInstance.methodname()
```

Custom Accessors

```
var balanceLessFees: Double
        get() {
                return accountBalance - fees
        }
        set(value) {
                accountBalance = value - fees
        }
```

#2. Kotlin Inheritance and Subclassing

Inheritance, Classes and Subclasses

Subclassing Syntax

parent:

```
open class MyParentClass {
        var myProperty: Int = 0
}
```

Children

```
class MySubClass : MyParentClass() {

}
```

#2. Kotlin Inheritance and Subclassing

Inheritance, Classes and Subclasses

Subclassing Syntax

parent:

```
open class MyParentClass {
        var myProperty: Int = 0
}
```

Children

```
class MySubClass : MyParentClass() {

}
```

Extending the Functionality of a Subclass

#2. Kotlin Inheritance and Subclassing

Extending the Functionality of a Subclass

```
class SavingsAccount : BankAccount {
    var interestRate: Double = 0.0

    constructor(accountNumber: Int, accountBalance: Double) :
super(accountNumber, accountBalance)

    fun calculateInterest(): Double
    {
        return interestRate * accountBalance
    }
}
```

#2. Kotlin Inheritance and Subclassing

Overriding Inherited Methods

```
class SavingsAccount : BankAccount {
        var interestRate: Double = 0.0
        constructor(accountNumber: Int, accountBalance: Double) :
        super(accountNumber, accountBalance)

        fun calculateInterest(): Double
        {
                return interestRate * accountBalance
        }
        override fun displayBalance()
        {
                println("Number $accountNumber")
                println("Current balance is $accountBalance")
                println("Prevailing interest rate is $interestRate")
        }
}
```

Adding a Custom Secondary Constructor

```
class SavingsAccount : BankAccount {

        var interestRate: Double = 0.0

        constructor(accountNumber: Int, accountBalance: Double) :
        super(accountNumber, accountBalance)

        constructor(accountNumber: Int, accountBalance: Double, rate: Double) :
super(accountNumber, accountBalance) {
        interestRate = rate
        }
}
```

#3. Kotlin Interfaces

- Interfaces in Kotlin can contain declarations of abstract methods, as well as method implementations

- Can have properties but these need to be abstract or to provide accessor implementations

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

#3. Kotlin Interfaces

Implementing Interfaces

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

Properties in Interfaces

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
    override val prop: Int = 29
}
```

#3. Kotlin Interfaces

Interfaces Inheritance

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

#4. Visibility Modifiers

The default visibility, used if there is no explicit modifier, is public

Packages

•If you do not specify any visibility modifier, public is used by default, which means that your declarations will be visible everywhere;
•If you mark a declaration private, it will only be visible inside the file containing the declaration;
•If you mark it internal, it is visible everywhere in the same module;
•protected is not available for top-level declarations.

#4. Visibility Modifiers

Classes and Interfaces

- Private means visible inside this class only (including all its members);
- Protected — same as private + visible in subclasses too;
- Internal — any client *inside this module* who sees the declaring -class sees its internal members;
- Public — any client who sees the declaring class sees its public members.

#5. Extensions

Kotlin provides the ability to extend a class with new functionality without having to inherit from the class

Extension functions

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}

val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'list’
```

Extension properties

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

#5. Extensions

Companion object extensions

```
class MyClass {
    companion object { }  // will be called "Companion"
}

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

#6. Data Classes

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

The compiler automatically derives the following members from all properties declared in the primary constructor:

- equals()/hashCode() pair;
- toString() of the form "User(name=John, age=42)";
- componentN() functions corresponding to the properties in their order of declaration;
- copy()

#6. Data Classes

Condition

- The primary constructor needs to have at least one parameter;
- All primary constructor parameters need to be marked as val or var;
- Data classes cannot be abstract, open, sealed or inner;

Properties Declared in the Class Body

```
data class Person(val name: String) {
    var age: Int = 0
}
```

Copying

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)

val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

Data Classes and Destructuring Declarations

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // prints "Jane, 35 years of age”
```

Standard Data Classes

The standard library provides Pair and Triple

Functions

1. Kotlin OOP – Class
2. Kotlin Inheritance and Subclassing
3. Kotlin Interfaces
4. Visibility Modifiers
5. Extensions
6. Data Classes

GLOBAL SMART TECHNOLOGIES

Kotlin Object Oriented Programming Advance Concept

#1. Generics

classes in Kotlin may have type parameters:

```
class Box<T>(t: T) {
    var value = t
}
val box: Box<Int> = Box<Int>(1)

val box = Box(1) // 1 has type Int, so the compiler figures out that we are talking about Box<Int>
```

#1. Generics

Generic functions

```
fun <T> singletonList(item: T): List<T> {
    // ...
}

fun <T> T.basicToString(): String {  // extension function
    // ...
}

val l = singletonList<Int>(1)
val l = singletonList(1)
```

Lesson Summary

- Kotlin Object Oriented Programming Basic concept
- Kotlin Object Oriented Programming Advance Concept

Thank you