5COSC023W - MOBILE APPLICATION DEVELOPMENT
Lecture 3: More on Kotlin - Anatomy of Composables

Dr Dimitris C. Dracopoulos

Classes

```
class Employee (colour: String, n: String) {
    val eyeColour: String = colour
    var age: Int = 25
    val name: String = n

    override fun toString(): String {
        return "name: $name, eyeColour: $eyeColour, age: $age"
    }
}

fun main() {
    val e1 = Employee("green", "John")
    println(e1)
}
```

Creating Class Properties Automatically

- Use `var` or `val` when you declare the parameters of the constructor:

```
class Employee (val eyeColour: String,
                var age: Int,
                var name: String) {
    override fun toString(): String {
        return "name: $name, eyeColour: $eyeColour, age: $age"
    }
}

fun main() {
    val e2 = Employee("brown", 18, "Helen")
    println(e2)
}
```

Secondary Constructors

Secondary constructors require the `constructor` keyword and they should be defined inside the curly braces of the class.

- Each secondary constructors needs to call directly or indirectly the primary constructor of the class using `this` keyword.

```
class Employee (var eyeColour: String,
                var age: Int,
                var name: String) {

    var salary = 0

    constructor (
                eyeColour: String,
                age: Int,
                name: String,
                sal: Int) : this(eyeColour, age, name) {
        salary = sal
    }
```

Secondary Constructors (cont'd)

```
    override fun toString(): String {
        return "name: $name, eyeColour: $eyeColour,
                age: $age, salary: $salary"
    }
}

fun main() {
    val e2 = Employee("brown", 18, "Helen", 40000)
    println(e2)
}
```

Data Classes

Classes which hold just data (not methods) can be created using *data classes*.

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

fun main() {
    var e1 = Employee("John", 22)
    var e2 = Employee("John", 22)

    println(e1 == e2)
}
```

Equality for data classes is automatically generated without defining the `equals` methods (which you need to define for the comparison of objects created from normal classes)

Default Values for Function Arguments

Function arguments can have an optional name and an optional default value.

- The order of arguments can be changed if their names is used.

```
fun colour(red: Int = 0, green: Int = 0, blue:Int = 0) {

}

fun main() {
    // default value for green is used, i.e. 0
    colour(blue = 255, red = 125)
}
```

Variable Number of Arguments

- Use the `vararg` keyword.
- The `vararg` parameter becomes an `Array`.
- A function definition can only specify one parameter as `vararg`.
- Try to choose the last parameter of a function to be the `vararg`.

```
fun foo(date: String, vararg names: String) {
    println("date: $date")
    for (n in names)
        println(n)
}

fun main() {
    foo("26th of February", "James", "Helen", "Joe", "Alice")
}
```

Lambda Expressions

Kotlin functions can be stored in variables, in data structures and passed as arguments to other functions.

- Lambda expressions and anonymous functions are function literals
- They can be treated as functions that are not declared but passed as an expression when a function is required.
- A lambda expression is always surrounded by curly braces.
- The body goes after the ->

```
// function foo accepts another function as an argument
fun foo(function_apply: (n: Int)->Int) {
    var x1 = function_apply(3)
    var x2 = function_apply(5)
    var x3 = function_apply(10)
    println("$x1, $x2, $x3")
}

fun main() {
    foo({n -> n*n})
    foo({n -> n + 1})
}
```

Passing lambdas as the last argument (Trailing Lambdas)

- If the last parameter of a function is a function, a lambda expression passed as an argument can be placed outside the parentheses

```
foo(){n -> n*2}
```

- If the lambda expression is the only argument to that call, the parentheses can be omitted:

```
foo{n -> n*2}
```

Lambdas with a Single Parameter

- If there is only 1 parameter in the lambda expression, the Kotlin compiler generates the name `it` for that parameter, which means that you can skip the need for "`n ->`":

```
fun foo(function_apply: (n: Int)->Int) {
    var x1 = function_apply(3)
    var x2 = function_apply(5)
    var x3 = function_apply(10)

    println("$x1, $x2, $x3")
}

fun main() {
    foo{it+2}  // add 2 to the passed parameter
    foo(i -> i+2)  // equivalent to the above
}
```

Maps

```
fun main() {
    var capitals = mapOf("Netherlands" to "Amsterdam",
                         "Hungary" to "Budapest",
                         "Finland" to "Helsinki")

    println(capitals["Hungary"])
    println(capitals.getValue("Finland"))

    for ((key, value) in capitals)
        println("$key -> $value")

    for (entry in capitals)
        println(entry.key + ":: " + entry.value)
}
```

Sets

Cannot contain duplicate elements.

```
fun main() {
  var cities = mutableSetOf("London", "Paris",
                            "Berlin", "London",
                            "Paris")

    for (c in cities)
        print(c+ " ")
    println()

    cities += "Warsaw"
    cities -= "Paris"

    print("Updated set contains: ")
    for (c in cities)
        print(c + " ")

}
```

The usual mathematical set operations (union, intersection,

Nullable References - An Attempt to fix Tony Hoare's "Billion Dollar Mistake"

- By default, references cannot receive the value of `null`.

```
var s: String = null  // Compiler error!
```

- A question mark ? needs to be appended to make a variable nullable:

```
var s: String? = null // OK
```

- A nullable type cannot be dereferenced:

```
var s2: String? = "abc"
s2.length  // Compiler error!
```

- Use the safe call `?.` to attempt to dereference a nullable value:

```
var s2: String? = "abc"
s2?.length  // Will give back a value of null if s2 is null
```

- Alternatively, use the non-null assertion operator `!!`

```
var s3: String? = "abc"
s3!!  // if null throws a NullPointerException
```

Comparing Variables

- Use == (or `equals`) for structural comparison
- Use === to check if 2 references point to the same object

For primitive types such as `Int`, === is the same as ==.

The When Expression

Similar to the switch in Java and other programming languages in the C family.

```
fun translate(word: String): String =
    when (word) {
        "Bonjour" -> "Good Morning"
        "Bonne Nuit" -> "Good Night"
        "Dobré Ráno" -> "Good Morning"
        "Dobrý Večer" -> "Good Evening"
        else -> "Unknown word"
    }

fun main() {
    var meaning = translate("Bonjour")
    println(meaning)
}
```

Access Specifiers

Similar usage to other programming languages supporting object oriented ptogramming but with different meaning.
When used for members (properties, functions) of a class:

- `public`: available to everyone
- `private`: available to the class only
- `protected`: subclasses can access and override these.
- `internal`: access only within the module where it is defined.

Default access is `public`.

- `public` and `private` can be used before the definition of a class, function or variable (property).

In such cases the meaning of `private` is access only within the same file.

Modules vs Packages

- Modules divide code at a higher level than packages.
- A library is often a single module consisting of multiple packages.
- The way a project is divided into modules, depends on the build system (e.g. gradle or maven).

The Anatomy of Composables

The Anatomy of Composables

- All composables are functions.

For example a `Button` defined in the library:

```
@Composable
public fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults. shape,
    colors: ButtonColors = ButtonDefaults. buttonColors(),
    elevation: ButtonElevation? = ButtonDefaults. buttonElevation(),
    border: BorderStroke? = null,
    contentPadding: PaddingValues = ButtonDefaults. ContentPadding,
    content: @Composable() (RowScope.() -> Unit)
): Unit
```

- Note that `content` is the last parameter passed to the `Button` composable.

The Anatomy of Composables

- All composables are functions.

For example a `Button` defined in the library:

```
@Composable
public fun Button(
    onClick: () -> Unit,
    modifier: Modifier = Modifier,
    enabled: Boolean = true,
    shape: Shape = ButtonDefaults. shape,
    colors: ButtonColors = ButtonDefaults. buttonColors(),
    elevation: ButtonElevation? = ButtonDefaults. buttonElevation(),
    border: BorderStroke? = null,
    contentPadding: PaddingValues = ButtonDefaults. ContentPadding,
    content: @Composable() (RowScope.() -> Unit)
): Unit
```

- Note that `content` is the last parameter passed to the `Button` composable.

The Anatomy of the Button Composable

Example of usage:

```
Button(onClick = {results = calculate()},
       modifier = Modifier.padding(top=10.dp),
       content = {
                  Text("Generate")
                  Text("Second text")
                 }
)
```

Using the **trailing lambda** technique, the above is equivalent to:

```
Button(onClick = {results = calculate()},
       modifier = Modifier.padding(top=10.dp)
      )
      {
         Text("Generate")
         Text("Second line")
      }
```