LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Praktikum Mobile und Verteilte Systeme

Kotlin 102

Prof. Dr. Claudia Linnhoff-Popien
Steffen Illium, Stefan Langer,
André Ebert
http://www.mobile.ifi.lmu.de
SoSe 2019

Google Blog
I/O 19 Announcement

Today we're announcing another big step: Android development will become increasingly Kotlin-first. Many new Jetpack APIs and features will be offered first in Kotlin. If you're starting a new project, you should write it in Kotlin; code written in Kotlin often mean much less code for you–less code to type, test, and maintain. And we're continuing to invest in tooling, docs, trainings and events to make Kotlin even easier to learn and use.

- Source:

https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html

Section 5: Functions

Content

```
Def:
    In programming, a named section of a program that
performs a specific task. In this sense, a function is
a type of procedure or routine.
```

- Defining and Calling Functions (fun x(){}; x())
- Default Arguments (var=0)

Functions
Defining and Calling Functions

```
fun functionName(para1: Type1,..., paraN: TypeN): Type {
    // Method Body
}
```

- Every function declaration starts with the **`fun`** keyword, has a function **name**, a list of **comma-separated parameters**, an optional **return type**, and a **method body**. The function parameters must be explicitly typed.

- Example:

```
fun avg(a: Double, b: Double): Double {
    return (a + b)/2
}
```

- Function call by name and passing of parameters

```
avg(4.6, 9.0) // 6.8
```

Functions
Single Expression Functions

- Return type and curly braces can be omitted if the function returns a single expression

```
// Declaration
fun avg(a: Double, b: Double) = (a + b)/2

//Call
avg(10.0, 20.0) // 15.0
```

- Return types are inferred by the compiler
- Functions which don't return anything has a return type of `Unit`. `Unit` corresponds to `void` in Java.
- Return Type Declaration of `Unit` is optional

```
fun printAverage(a: Double, b: Double) {
    println("Avg of ($a, $b) = ${(a + b)/2}")
}
```

Functions
Default Arguments

- Specification of default values for function parameters

```
fun display(message: String, name: String = „Student") {
    println("Hello $name, $message")
}
```

```
Display%("welcome to MSP")
// Hello Student, welcome to MSP
```

- Possible collision with positional arguments

Functions
Named Arguments

- More readable function calls

```
displayGreeting(message="welcome to MSP")
// Hello Student, welcome to MSP
```

- Passing of parameter values selectively if other parameters have default values

```
fun display(message: String = "i hope you like it",
            name: String) {
    println("Hello $name, $message")
}
```

- Positional arguments can be skipped with named passing
- Positional arguments are placed before the named arguments

```
displayGreeting("Yorik", message="welcome to MSP")
// Hello Yorik, welcome to MSP
```

Functions
Variable Number of Arguments

- Passing a variable number of arguments to a function by declaring the function with a `vararg` parameter
- A `vararg` parameter is internally represented as `Array<T>`

```
fun sum(vararg numbers: Double): Double {
    var sum: Double = 0.0
    for(number in numbers) {
        sum += number
    }
    return sum
}
```

```
sum(1.5, 2.0)
// Result = 3.5

sum(1.5, 2.0, 3.5, 4.0, 5.8, 6.2)
// Result = 23.0
```

Functions
Variable Number of Arguments II

- A function may have only one `vararg` parameter.
- Other parameters following the `vararg` parameter are passed using the named argument syntax

```
fun sum (vararg numbers: Double,
         initialSum: Double = 0.0): Double {
    var sum = initialSum
    for(number in numbers) {
        sum += number
    }
    return sum
}
```

```
sum(1.5, 2.5, initialSum=100.0) // Result = 104.0
```

- It is possible to pass a array using the `spread` operator

```
val a = doubleArrayOf(1.5, 2.6, 5.4)
sum(*a) // Result = 9.5
```

Functions
Infix Notation

- Transformation of function which require a single argument
- Omitting of braces and bodies
- It is characterized by the placement of operators between operands

- Example: the plus sign in 2 + 2.

```
val c1 = 5.0
val c2 = 7.0

// Usual call
c1.add(c2)

// Infix call
c1 add c2

// produces - 12.0
```

More Information:

callicoder.com/kotlin-infix-notation

Section 6: OOP
Content

```
Def:
    Object-oriented programming (OOP) is a programming
paradigm based on the concept of "objects", which can
contain data, in the form of attributes, and code, in
the form of methods. A class is a blueprint for
creating objects of similar type.
```

- Classes and Objects
  - Getters and Setters
  - Inheritance
- Abstract Classes
- Data Classes

OOP
Classes

- Creating classes using the `class` keyword

```
class Person {
}
```

- … or without curly braces

```
class Person
```

- Initialization using the default constructor:

```
val person = Person()
```

- … there is no `new` keyword in Kotlin.

OOP
Properties

- Property declaration (member variables) using `var` / `val` keyword

```
class Person {
    // Properties or Member Variables
    var name: String
}
```

- Does not compile: member Variables must be initialized!
- This can be solved by Default Values

```
class Person {
    var firstName: String = "Guest"
    var lastName: String = ""
}
```

- … or constructors

OOP
Primary Constructors I

- Primary Constructors as part of the class header

```
class Person (firstName: String, lastName: String) {
}
```

- List of **comma-separated parameters** with **type** modifiers
- `constructor` keyword if class has visible modifiers or annotations (like public, private, protected)

```
class Person constructor (firstName: String) {
}
```

- Initialization Logic is written within the **initializer block**

- … or directly declare them in class body itself.

OOP
Primary Constructors II

```
class Person(_firstName: String, _lastName: String) {
    // Member Variables (Properties) of the class
    var firstName: String
    var lastName: String

    // Initializer Block
    init {
        this.firstName = _firstName
        this.lastName = _lastName

        println("Initialized a new Person object:")
        println("firstName = $firstName")
        println("lastName = $lastName")
    }
}
```

OOP
Primary Constructors III

```
class Person(_firstName: String, _lastName: String) {
    // Member Variables of the class
     var firstName = _firstName
     var lastName = _lastName
    // Initializer Block
    init {
        println("Initialized a new Person object:")
        println("firstName = $firstName")
        println("lastName = $lastName")
    }
}
```

OOP
Primary Constructors IV

- Using concise syntax for declaration and initialization of properties
- This can also include default values

```
class Person(var firstName: String = "Guest",
             var lastName: String = "") {
    // Initializer Block
    init {
        println("Initialized a new Person object:")
        println("firstName = $firstName")
        println("lastName = $lastName")
    }
}
```

- Object Creation with classes having default parameters

```
val person1 = Person("Jack", "Dorsey")
val person2 = Person("Jack")
val person3 = Person()
```

OOP
Modifiers

- Visibility modifiers help us restrict the accessibility of classes, objects, constructors, functions or properties.

- There are four types of visibility modifiers in Kotlin:
  - `public` - Anything that is declared public is **accessible everywhere**.

  - `private` - A top-level function or class that is declared private can be accessed **only within the file** where it is declared.

    Any member function, constructor, or property that is declared private is visible **only within the class** where it is declared.

  - `protected` - Any property or function declared as protected is accessible in the same class and its **subclasses**

- The default visibility for everything is public

OOP
Getters and Setters

- **No** need to define getters and setters for properties of an object

- Kotlin **automatically** generates an implicit getter and a setter for mutable properties, and a getter (only) of read-only properties.

```
val person = Person("Sundar", "Pichai")
println(person.firstName) // Sundar

person.lastName = "Jobs"
println("Name = ${person.firstName} ${person.lastName}")
// Sundar Jobs
```

OOP
Getters and Setters II

- Example for public class creation with read-only properties:

```
class Student (val rollNumber: Int, val name: String)

val student = Student(1, "John")

println(student.rollNumber)
println(student.name)

student.name = "Jack"
// Error: Val can not be assigned
```

OOP
Custom Getters and Setters

- Default definition of getters and setters.
- Notice the correct usage of `value` and `field`

```
class User(_id: Int, _name: String, _age: Int) {
    val id: Int = _id
        get() = field

    var name: String = _name
        get() = field
        set(value) {
            field = value
        }
}
```

- `Field` and `value` can be renamed

OOP
Custom Getters and Setters II

```
class User(_id: Int, _name: String, _age: Int) {
    val id: Int = _id
    var name: String = _name
    // Custom Getter
    get() {
        return field.toUpperCase()
    }
    var age: Int = _age
    // Custom Setter
    set(value) {
        field = if(value > 0) value else throw Illegal…
    }
}

val user = User(1, "Jack Sparrow", 44)
user.age = -1
// Throws IllegalArgumentException: Age to low
```

OOP
Inheritance

- All the classes in Kotlin have a common base class called `Any`
- Every class that you create in Kotlin implicitly inherits from `Any`
- There are three methods which are always implemented:

```
package kotlin

/**
 * The root of the Kotlin class hierarchy. Every Kotlin class has [Any] as a superclass.
 */
public open class Any {

    public open operator fun equals(other: Any?): Boolean


    public open fun hashCode(): Int


    public open fun toString(): String
}
```

OOP
Base and Derived classes

- Base classes have to be decorated with the `open` modifier
- Child classes have to initialize the parent class.

```
// Base class (Super class)
open class Computer { }

// Derived class (Sub class)
class Laptop: Computer() { }
```

- If the child class has a primary constructor, parent parameters must initialized in the class

```
// Child class (initializes the parent class)
class Laptop(name: String,
             brand: String,
             val batteryLife: Double) : Computer(name, brand) {
    // Class Body
 }
```

OOP
Overriding Member Functions

- Like classes, its members are also final by default.
- `open` modifier allows to override them

- Moreover, a child class has to use the `override` modifier to override a base class member

```
open class Teacher {
    open fun teach() {
        println("Teaching...")
    }
}

class MathsTeacher : Teacher() {
    override fun teach() {
        println("Teaching Maths...")
    }
}
```

OOP
Calling Super Properties

- Parent members are shadowed by the child class implementation
- Access possible by use of `super()` keyword in the child class

```
open class Employee {

    open val salary: Double = 10000.0

}

class Developer: Employee() {

    override var salary = super.salary + 10000.0

}
```

OOP
Abstract Classes

- Declaration of abstract classes with the `abstract` keyword
- Explicit use of `abstract` keyword for abstract class members
- Any subclass that extends the abstract class must implement all of its abstract methods and properties, or the subclass should also be declared as abstract
- Abstract classes are `open` to be overridden by default
- The same applies for abstract class members
- Overriding non abstract class members again need the `open` modifier

OOP
Abstract Classes

- Example:

```
abstract class Vehicle(val name: String,
                       val color: String,
                       val weight: Double) {

    // Abstract Property
    abstract var maxSpeed: Double

    // Abstract Method
    abstract fun start()
    abstract fun stop()

    // Concrete (Non Abstract) Method
    open fun displayDetails() {
        println("Name: $name, Color: $color")
    }
}
```

Functions
Importing Package Functions

- Additional package member functions can be imported.

- They are referred to by the dot . Notation

```
import foo.Bar
// Bar is now accessible without qualification
```

- The import keyword is not restricted to importing classes; you can also use it to import other declarations:

```
// Bar is accessible
import foo.Bar

// bBar stands for 'bar.Bar'
import bar.Bar as bBar
```

Functions
Data Classes

- Data Classes automatically implement the standard boilerplate functions like `getters`, `setters`, `equals`, `toString` and `hash` functionality

```
data class Customer(val id: Long, val name: String)
```

- equals():

```
val customer1 = Customer(1, "John")
val customer2 = Customer(1, "John")

// Prints true
println(customer1.equals(customer2))

// Same as
println(customer1 == customer2)
```

Functions
Data Classes II

- toString():

```
val customer = Customer(2, "Robert")

println("Customer Details : $customer")

// Customer Details : Customer(id=2, name=Robert)
```

- copy():

```
val updatedCustomer = customer.copy(name = "James")

println("Customer : $customer")
println("Updated Customer : $updatedCustomer")
```

```
// Customer : Customer(id=2, name=Robert)
// Updated Customer : Customer(id=2, name=James)
```

Functions
Data Classe: Destructing Declarations

- Generation of componentN() functions corresponding to all the properties declared in the primary constructor of the data class.

```
val customer = Customer(4, "Joseph")

println(customer.component1())
// Prints 4
println(customer.component2())
// Prints "Joseph"
```

- The **Destructuring declaration** syntax helps you destructure an object into a number of variables

```
val customer = Customer(4, "Joseph")

// Destructuring Declaration
val (id, name) = customer
println("id = $id, name = $name")
// Prints "id = 4, name = Joseph"
```