Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Basics of Mobile Application Development

Kotlin

Kotlin

History

- The name comes from Kotlin Island, near St. Petersburg.
- July 2011 - JetBrains unveiled Project Kotlin
  - One of the stated goals of Kotlin is to compile as quickly as Java.
- February 2012 - JetBrains open sourced the project under the Apache 2 license
- Kotlin v1.0 was released on February 15, 2016
- At Google I/O 2017, Google announced first-class support for Kotlin on Android
- Kotlin v1.2 was released on November 28, 2017
- Kotlin v1.3.5 was released on August 22, 2019

Features

- Statically typed programming language
- Runs on the Java virtual machine
  - can be compiled to JavaScript source
- Syntax is not compatible with Java
  - The JVM implementation of the Kotlin standard library interoperates with Java code
  - Relies on Java code from the existing Java Class Library
- Uses aggressive type inference to determine the types of values and expressions for which type has been left unstated.
- Kotlin code can run on JVM up to latest Java 11.
- As of Android Studio 3.0, Kotlin is fully supported by
  - The Android Kotlin compiler lets the user choose between targeting Java 6, or Java 8-compatible bytecode.

Philosophy

- Kotlin is designed to be an industrial-strength object-oriented language
  - A "better language" than Java, but still be fully interoperable with Java code
- Semicolons are optional as a statement terminator
  - in most cases a newline is sufficient for the compiler to deduce that the statement has ended
- Kotlin variable declarations and parameter lists have the data type come after the variable name (and with a colon separator), similar to Pascal.
- Variables in Kotlin can be immutable, declared with the val keyword, or mutable, declared with the var keyword.
- Class members are public by default, and classes themselves are sealed by default, meaning that creating a derived class is disabled unless the base class is declared with the open keyword.
- In addition to the classes and methods of object-oriented programming, Kotlin also supports procedural programming with the use of functions.

Variables – immutable

- Immediate assignment
  - `val a: Int = 1`
- Int type is inferred
  - `val b = 2`
- Type required when no initializer is provided
  - `val c: Int`
- Deferred assignment
  - `c = 3`

Variables – mutable

- Int type is inferred
  - `var x = 5`
  - `x += 1`
- Value assigment

Basic types

- Numbers are similar as in Java, but not exactly the same.
  - There are no implicit widening conversions for numbers, and literals are slightly different in some cases.
- Built-in types for numbers
  - Double 64 bit
  - Float 32 bit
  - Long 64 bit
  - Int 32 bit
  - Short 16 bit
  - Byte 8 bit

Literals

- Decimals: `123`
  - Longs are tagged by a capital L: `123L`
- Hexadecimals: `0x0F`
  - `Binaries: 0b00001011`
- Underscore
  - `val oneMillion = 1_000_000`
  - `val hexBytes = 0xFF_EC_DE_5E`

Basic types

- Characters are represented by the type Char
- The type Boolean represents booleans, and has two values: true and false
- Arrays are represented by the Array class
  - get and set functions
  - size property
- Strings are represented by String.
  - Immutable
  - Elements of a string are characters that can be accessed by the indexing operation: s[i]

String template

- Strings may contain template expressions
  - A template expression starts with a dollar sign ($) and consists of either a simple name:
    - `val i = 10`
    - `println("i = $i") // prints "i = 10"`
  - val s = "abc"
  - println("$s.length is ${s.length}") // prints "abc.length is 3"

Range

- Defined by ..

```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```

- Skip numbers
  - `1..10 step 2`
- Reverse
  - `9 downTo 0 step 3`

Collections

- List
  - `val fruits = listOf("banana", "avocado", "apple", "kiwifruit")`

Functions

- Declaring a function (not only as part of a class!)

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

- As expression

```
fun sum(a: Int, b: Int) = a + b
```

- Method

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

`: Unit` is optional

Nested functions

```
fun saveUser(user: User) {
 fun validate(user: User, value: String, fieldName: String) {
  if (value.isEmpty()) {
   throw IllegalArgumentException("Can't save user")
  }
 }
 validate(user, user.name, "Name")
 validate(user, user.address, "Address")
}
```

Named Arguments

- Function parameters can be named when calling functions

```
fun reformat(str: String,
             normalizeCase: Boolean = true,
             upperCaseFirstLetter: Boolean = true,
             wordSeparator: Char = ' ') {
                ...
             }
reformat(str)
reformat(str, true, true, '_')
reformat(str,
    normalizeCase = true,
    upperCaseFirstLetter = true,
    wordSeparator = '_'
)
```

Infix notation

- Functions marked with the infix keyword can also be called using the infix notation

```
infix fun Int.shl(x: Int): Int { ... }

1 shl 2
// is the same as
1.shl(2)
```

Control statements

- Conditional statement

```
if (a > b) {
        // do a
} else {
        // do b
}
```

- You can use if as an expression

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

Control statements

- Conditional statement as an expression

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b

val max = if (a > b) {
    print("Choose a")
    a
} else {
    print("Choose b")
    b
}
```

Control statements

- When (instead of switch)

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

Control statements

- When, multiple cases

```
when (x) {
    0, 1 -> print("x == 0 or x == 1")
    else -> print("otherwise")
}
```

- Expressions

```
when (x) {
    parseInt(s) -> print("s encodes x")
    else -> print("s does not encode x")
}
```

Control statements

- Intervals

```
when (x) {
    in 1..10 -> print("x is in the range")
    in validNumbers -> print("x is valid")
    !in 10..20 -> print("x is not in the range")
    else -> print("none of the above")
}
```

- When and functions

```
fun hasPrefix(x: Any) = when(x) {
    is String -> x.startsWith("prefix")
    else -> false
}
```

Control statements

- For loop

```
for (item in collection) print(item)

for (i in 1..3) {
    println(i)
}

for (i in 6 downTo 0 step 2) {
    println(i)
}
```

Control statements

- While loop

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

Control statements

- Unconditional
  - `return`: By default returns from the nearest enclosing function or anonymous function.
  - `break`: Terminates the nearest enclosing loop.
  - `continue`: Proceeds to the next step of the nearest enclosing loop.
- Labels
  - Any expression in Kotlin may be marked with a label and can be used with break and continue

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (...) break@loop
    }
}
```

Null-safety

- Error:
  - `var a: String = "abc"`
  - `a = null`
- Allowed to be null
  - `var b: String? = "abc"`
  - `b = null`
  - `print(b)`

Null-safety

- Checking
  - `val l = if (b != null) b.length else -1`
  - `val l = b?.length ?: -1`
  - `println(b?.length)`
- Assertation
  - `val l = b!!.length`
- Safe cast
  - `val aInt: Int? = a as? Int`

Type safety and casts

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        return obj.length
    }
     return null
}
```

Classes

- Class declaration
  - `class Car { ... }`
- It can be empty
  - `class Empty`
- Creating an instance
  - `val car = Car()`
  - `val customer = Customer("Joe Smith")`

Properties

- Example

```
class Address {
    var name: String = ...
    var street: String = ...
    var city: String = ...
    var state: String? = ...
    var zip: String = ...
}
```

Get and set

- Defaults
  - explicit initializer required, default getter and setter implied
    - `var allByDefault: Int? // error`
  - default getter and setter
    - `var initialized = 1`
  - default getter, must be initialized in constructor
    - `val simple: Int?`
  - default getter
    - `val inferredType = 1`

Get and set

- Custom get

```
val isEmpty: Boolean
    get() = this.size == 0
```

- Alternative get

```
val isEmpty get() = this.size == 0
```

- Get and set

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value)
    }
```

Fields

- Fields cannot be declared directly in Kotlin classes
  - When a property needs a backing field, Kotlin provides it automatically
- This backing field can be referenced
  - The `field` identifier can only be used in the accessors of the property.

```
var counter = 0
    set(value) {
        if (value >= 0) field = value
    }
```

- No backing field

```
val isEmpty: Boolean
    get() = this.size == 0
```

Constructors

- Primary constructor (optional)
  - Part of the class header
  - Cannot contain any code
  - Receives parameters that can be used
    - For property initialization
    - Or inside init blocks
  - `class Person constructor(firstName: String) { ... }`
  - If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted
    - `class Person(firstName: String) { ... }`

Initialization

- The primary constructor cannot contain any code.
- Initialization code can be placed in initializer blocks.
- During an instance initialization, the initializer blocks are executed in the same order as they appear in the class body, interleaved with the property initializers:

```
class InitOrderDemo(name: String) {
    init {
        println("Accessing the field $name")
    }
    val nameProperty: String
    init {
        println("Setting to a property")
        nameProperty=name
    }
}
```

Primary constructor and initialization

- For declaring properties and initializing them the primary constructor can be used as follows:

```
class Person(val firstName: String, val
lastName: String, var age: Int)
 { … }
```

  - Much the same way as regular properties, the properties declared in the primary constructor can be mutable (var) or read-only (val).
- Also you can call functions on that parameters

```
class Customer(name: String) {
    val customerKey = name.toUpperCase()
}
```

Constructors – Secondary

- Secondary constructor(s)

```
class Person {
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

- If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor
  - either directly or indirectly

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Inheritance

- All classes in Kotlin have a common superclass Any
  - `class Example`
  - Any is not `java.lang.Object;`
  - Members
    - `equals()`
    - `hashCode()`
    - `toString()`
- Classes are „closed" in default
  - Cannot inherit from them
  - Thus we need to „open" to create children classes

Inheritance

- Example
  - `open class Base(p: Int)`
  - `class Derived(p: Int) : Base(p)`

Constructor

- If the derived class has a primary constructor, the base class must be initialized right there
  - using the parameters of the primary constructor
- If the class has no primary constructor
  - then each secondary constructor has to initialize the base type using the super keyword
  - or to delegate to another constructor which does that

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)
    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Overriding

- Kotlin requires explicit modifiers for overridable members

```
open class Base {
    open fun v() { ... }
    fun nv() { ... }
}
class Derived() : Base() {
    override fun v() { ... }
}
```

Overriding

- A member marked override is itself open
  - It may be overridden in subclasses
  - If you want to prohibit re-overriding, use final

```
open class AnotherDerived() : Base() {
    final override fun v() { ... }
}
```

Properties

- Overriding properties works in a similar way to overriding methods

```
open class Foo {
    open val x: Int get() { ... }
}

class Bar1 : Foo() {
    override val x: Int = ...
}
```

Calling the superclass implementation

```
open class Foo {
    open fun f() { println("Foo.f()") }
    open val x: Int get() = 1
}
class Bar : Foo() {
    override fun f() {
        super.f()
        println("Bar.f()")
    }
    override val x: Int get() = super.x + 1
}
```

Abstract class

- A class and some of its members may be declared abstract
- An abstract member does not have an implementation in its class

```
open class Base {
    open fun f() {}
}

abstract class Derived : Base() {
    override abstract fun f()
}
```

Interfaces

- Interfaces are very similar to Java 8

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

- Implementation

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

Properties

- You can declare properties in interfaces.
  - A property declared in an interface can either be abstract, or it can provide implementations for accessors.
  - Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them

```
interface MyInterface {
    val prop: Int // abstract
    val propertyWithImplementation: String
        get() = "foo"
    fun foo() {
        print(prop)
    }
}
class Child : MyInterface {
    override val prop: Int = 29
}
```

Interface inheritance

- An interface can derive from other interfaces and thus both provide implementations for their members and declare new functions and properties

```
interface Named {
    val name: String
}

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

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

Multiple inheritance

```
interface A {
    fun foo() { print("A") }
    fun bar()
}
interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}
class C : A {
    override fun bar() {
      print("bar")
    }
}
```

```
class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }
    override fun bar() {
        super<B>.bar()
    }
}
```

Visibility

- Classes, objects, interfaces, constructors, functions, properties and their setters can have visibility modifiers.
  - Getters always have the same visibility as the property.
- There are four visibility modifiers
  - private, protected, internal,public
- The default visibility is public.

Visibility

- Explanation
  - 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.

Visibility

- Classes and Interfaces
  - private: 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

Packages – Modules

- Packages
  - Functions, properties and classes, objects and interfaces can be declared directly inside a package

```
package foo
fun baz() { ... }
class Bar { ... }
```

- Modules
  - A set of Kotlin files compiled together:
    - an IntelliJ IDEA module;
    - a Maven project;
    - a Gradle source set (with the exception that the test source set can access the internal declarations of main);
    - a set of files compiled with one invocation of the <kotlinc> Ant task

Extension

- To extend a class with new functionality without having to inherit from the class
- Or use any type of design pattern such as Decorator

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

val l = mutableListOf(1, 2, 3)
l.swap(0, 2)
```

Extension

- Extension properties

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

Data classes

- To create classes whose main purpose is to hold data
  - 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() function

Data classes

- Data classes have to fulfill the following requirements:
  - 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.

Enum

- The most basic usage of enum classes is implementing type-safe enums:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}

enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },
    TALKING {
        override fun signal() = WAITING
    };
    abstract fun signal(): ProtocolState
}
```

Delegation

- Supported natively:
  - A class Derived can implement an interface Base by delegating all of its public members to a specified object:

```
interface Base {
    fun print()
}
class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}
class Derived(b: Base) : Base by b
fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

Generics

- As in Java, classes in Kotlin may have type parameters:

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

- Functions can have type parameters

```
fun <T> singletonList(item: T): List<T> {
    // ...
}
val l = singletonList<Int>(1)
```

Java <-> Kotlin

- Existing Java code can be called from Kotlin in a natural way

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

Java <-> Kotlin

- Existing Java code can be called from Kotlin in a natural way
  - Escaping

foo.`is` (bar)

Java <-> Kotlin

- Kotlin is called in Java (smooth)
  - A Kotlin property is compiled to the following Java elements:
    - A getter method, with the name calculated by prepending the get prefix;
    - A setter method, with the name calculated by prepending the set prefix (only for var properties);
    - A private field, with the same name as the property name (only for properties with backing fields).
  - Package-Level Functions
    - All the functions and properties declared in a file example.kt inside a package org.foo.bar, including extension functions, are compiled into static methods of a Java class named org.foo.bar.ExampleKt

Java <-> Kotlin

- Kotlin is called in Java (smooth)
  - The Kotlin visibilities are mapped to Java in the following way:
    - private members are compiled to private members;
    - private top-level declarations are compiled to package-local declarations;
    - protected remains protected (note that Java allows accessing protected members from other classes in the same package and Kotlin doesn't, so Java classes will have broader access to the code);
    - internal declarations become public in Java;
    - public remains public.

Homework – Deadline 11/26 10.15 am

- You have to create a demonstration of Kotlin object oriented capabilities
- Details
  - Create a class to represent any cards
    - Content, initialization
    - Comparison
  - Create a class to represent playing cards
    - Suit, rank
    - Use inheritance
    - Use enum
  - Create two classes to store deck for cards and playing cards as well
- Test your code

Android Kotlin

Next week