CS 478

Software Development for Mobile Platforms

Set 8: Introduction to Kotlin

Ugo Buy
Department of Computer Science
University of Illinois at Chicago

April 30, 2019

0

Kotlin

- Programming language designed by Dmitry Jemerov and his team at JetBrains
- Named after island near St. Petersburgh, Russia
- Project Kotlin announced in July 2011
- Kotlin 1.0 released on February 15, 2016
- Full Android support announced at Google I/O 2017 (May 2017)
- Latest version: 1.3.30 (April 12, 2019)

1

Kotlin architecture

- Fully compatible with Java Virtual Machine
  - Compile to Java 1.6 or 1.8 bytecodes (your choice)
  - Compatible also with LLVM language system, and with JavaScript
- Fully Java interoperable

  Support for:
  - Call-out to Java classes and functions from Kotlin code
  - Call-in from Java code to Kotlin code
  - **Hybrid OO language:** Standalone functions alongside classes and methods (Java is pure OO)
  - **Statically typed:** Identifiers have unique data type, available at compile-time

2

Kotlin overview: Type system

- Statically typed
  - Identifiers have unique data type, available at compile-time
- Strong type inferencing
  - Identifier's type need not be specified by programmer; can be inferred from value assignment
- Nullable vs. non-nullable data types
  - Avoid *NullPointerExceptions (NPEs)* at run-time
  - Catering to distracted programmers?
- The "retro" syntax:
  - Variables definitions resemble Pascal
  - Function definitions and function calls resemble Smalltalk

3

Kotlin overview: Syntax

- Semicolons optional as statement terminators
  - New lines will do too
- Class fields and methods are public (not package accessible) by default
  - This is probably a bad idea for fields, no information hiding
- Classes are **final** by default
  - Must declare them *open* to enable subclassing
  - But: What is the point of inheritance if you disable it by default?
  - The best part: Class methods are dispatched dynamically (aka message polymorphism) as if to assume inheritance will happen
- No *static* class fields and methods (Use file-scope definitions instead)
  - Why is this a bad idea: Loss of scope information

4

Kotlin overview (cont'd)

- Significant omissions:
  - No language support for concurrency (e.g., threads)
  - No language-defined IPC (locking, message passing), use OS primitives
  - No class (*static)* fields (just like in Objective C and Swift) ☹
- Weird object model – Inheritance not supported by default, but dynamic message dispatching is
- Backward compatibility across versions?

5

Kotlin lineage

- Clearly derived from Java
- But strong similarity to Swift (Apple's new language, since 2014)

Kotlin vs. Swift

- See what Kotlin features match ✔ Swift and what features don't ✗
  - Object-oriented but hybrid (supporting procedural paradigm too) ✔
  - Statically typed but with type inference ✔
  - Single inheritance with protocols—Kotlin has interfaces ✔
  - No root superclass! (Kotlin root class is called *Any)* ✗
  - No implicit type conversions for numeric types ✔
  - Non garbage-collected but with automatic reference counting ✗
  - Support for block closures ✔
  - Support for exception handling since Swift V1.2 ✔
  - Classes _and_ structs (called *data classes* in Kotlin) ✔

Reading Materials

- Online documentation and tutorials on Kotlin:
  - ➢ Tutorial point Kotlin tutorial:
    https://www.tutorialspoint.com/kotlin/index.htm
  - ➢ Kotlin language's official web site
    https://kotlinlang.org/
  - ➢ Kotlin for Android
    https://kotlinlang.org/docs/reference/android-overview.html
  - ➢ Android official web page
    https://developer.android.com/kotlin/
  - ➢ Lynda tutorial (freely available to anyone with a UIC NetID)
    https://www.lynda.com ...

Outline of Kotlin topics

- IDE support for Kotlin
- Kotlin literals
- Type system
- Scope rules and block closures
- Flow of control
- Classes
- Inheritance

IDE support

- Eclipse users: Kotlin Plug-in
  - In Eclipse *Help → Eclipse Marketplace ...*
  - Filter "Kotlin" and install resulting IDE

10

IDE support

- IntelliJ users: Support from *Jet Brains IntelliJ Idea* IDE
  - Basis of Android Studio IDE that we will use for this course
  - Automatic translation (incomplete) from Java to Kotlin
  - But better off writing apps directly in Kotlin
- Course instructor used Eclipse for learning language and Android Studio for writing apps

11

Kotlin: Literals

Kotlin literals: Numbers

- Support for decimal, hexadecimal and binary integers (no octal), e.g.,
  - Decimal: 99, 99L (99 as a long)
  - Hex: 0x63
  - Binary: 0b01100011
- Floating point literals are *Double* by default, use
  - A *Double* number: 42.5
  - *Float* literals: 42.5f – 42.5F
- Underscores are allowed!
  - 10_000_000.5 (more legible?)

Kotlin literals: Strings

- Basic strings use the usual double quote notation
  - **"Hello there"**
- Like Java's, Kotlin strings are immutable and possibly escaped
  - **"Hello", " there\n"**
- New concept: *Raw strings*
  - Not escapable, content taken literally including white space and newline characters
  - Syntax: Delimited by a triple double quote character sequence
  - **"""**
    Roses are red
    Violets are blue
    Honey is sweet
    And so are you (Joseph Ritson, 1784) """

14

String templates

- Embed expression evaluation in string
- Syntax: `$token` or `${expression}`
- Examples
  - **"The value of variable x is $x."**
  - **"The sum of x and y is ${x+y}."**

15

More literals

Data types with an explicit representation

- Logical: **true**, **false**
- Characters: **'a'**, **'5'**, **'$'** (single quotes)
- Ranges: **x..y**, **1..5** (double dots, inclusive at both ends)
- Indexing: **anArray[i]** (square brackets)
- Comments: As usual
  - (1) Multiline **/* … */**
  - (2) End-of-line **// …**
- Reference: https://www.tutorialspoint.com/kotlin/kotlin_quick_guide.htm

16

Kotlin: Type System

17

Kotlin's type system

- Statically typed language with type inference
- Kotlin type system = Basic types + Class types
- Basic types mirror Java's *primitive* types (e.g., **int, boolean, char, double**)
- Basic types use value (copying) semantics (e.g., for assignment, etc.)
- Class types use reference (non-copying) semantics
- Also, *nullable* vs. *Non Nullable* (default) versions of each data type
  - Nullable types are always references
  - Similar to Java: **int** is not nullable but **Integer** is nullable
  - But now you have declare upfront whether variable is nullable

18

Kotlin's type system

Basic data types

- Numeric: **Byte** (8 bits), **Short** (16), **Int** (32), **Long** (64), **Float** (32), **Double** (64)
- Logical: **Boolean**
- Characters: **Char, String** (array of **Char**)
- Arrays: **ByteArray**, **CharArray, ShortArray**, **IntArray**, **LongArray**, **FloatArray**, **DoubleArray**

19

Kotlin's variable definitions

- Syntax of variable and constant identifiers:
  - **Variable definitions**: Keyword **var** followed by identifier, colon, type declaration, (optional) equal sign and initial value

    var <var_id>: <type_id> [= <init_value>]
  - Example:

    var i: **Int** = 18
  - **Constant definitions**: Keyword **val** followed by identifier, colon and type declaration (and possibly constant value)

    val <var_id>: <type_id> [= <init_value>]
  - Example:

    val j: **Int** = 18

20

Kotlin variable definitions (cont'd)

- Notice syntax of function definition
- Type of string constant *s1* is inferred

```
fun main(args: Array<String>) {
   val i: Int   // Defining a constant
   var j: Int   // Defining a non-nullable variable
   i = 10       // Do this only once
   j = 20       // OK, assigning a value to a variable

   // Defining a constant, using type inferencing
   val s1 = "Hello there"
   println("The second element of s1 is: " + s1[1] + ".\n”)
}
```

21

Conversions

- Conversion: Creation of a new instance, based on existing instance
  - Convert 100 to 100L
  - Performed at run-time, some languages support automatic conversions
- No automatic conversions for numeric types, must convert explicitly
  - **toByte(): Byte**
  - **toShort(): Short**
  - **toInt(): Int**
  - **toLong(): Long**
  - **toFloat(): Float**
  - **toDouble(): Double**
  - **toChar(): Char**

22

Examples of conversions

- Implicit conversion does not happen, explicit conversion is OK

```
fun main(args: Array<String>) {

   val i: Byte = -100  // Defining a Byte variable

   var j: Int = i      // C-T error, no implicit conversion
                       // Byte → Int

   var k: Int = i.toInt()
                       // OK, explicit conversion, anonymous
                       // Int created from Byte
}
```

23

Conversions (cont'd)

- Arithmetic operations are overloaded for mixed-type operands
  - Sum operator '+' works on operands of type Byte, Short, Float, etc.
  - But can also mix integral and floating point types
  - E.g., add a long and a byte to produce a long
  - Make absence of implicit conversions less bothersome

OK examples.

```
var aFloat: Float
var aDouble: Double
aFloat = 1.0F
aDouble = 2.0 + aFloat   // OK, now aDouble == 3.0
aDouble = 12.0 + 3       // still OK, now aDouble == 15.0
aFloat = 3.0             // not OK, 3.0 is a Double literal
```

Compile-time error.

24

Type Inference

- Kotlin can figure out identifier's type from context where identifier defined
- No need to declare identifier's type explicitly in this case
- The following ...

Explicit type declarations.

```
var x: Int      // Defining a variable identifier
val y: Int      // Defining a constant identifier
x = 10          // Assign the variable
y = x + 5       // Set constant value: Do this only once!
```

… is equivalent to:

Type implicitly inferred from initializer.

```
var x = 10      // Inferring x's Int type from initializer
val y = x + 5   // Inferring y's Int type from initializer
```

25

Nullable data types

- Data types that can take the special value **`null`**
  - **`null`** is the well-known null pointer, assignable to any identifier
- By default, any data type must be assigned some value different from **`null`**
  - But nullable types can be assigned **`null`**
- Syntax: Put question mark after data type, to make it nullable, e.g.,
  - `var x: Int? = 10`
- Examples

```
var x: Int = 10       // Non null data type
val y: Int? = 20      // Nullable data type
y = null              // OK
x = null              // Compiler error!
```

26

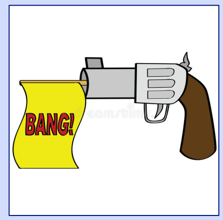
Handling *nullable* data types

- Variables of a *nullable* type cannot be accessed directly, because they could be **`null`**
- Unchecked access to *nullable* variable returns a C-T error! ☹
- https://kotlinlang.org/docs/reference/null-safety.html

27

Handling *nullable* data types (cont'd)

- Ways to access *nullable* variable:
  1. Wrap access in **`null`** check
  2. Safe calls, **`?.`** operator
  3. Elvis, **`?:`** operator
  4. Bang, bang, **`!!`** operator
  5. Safe cast
- https://kotlinlang.org/docs/reference/null-safety.html



28

1. Wrap access

- Wrap access to *nullable* variable in `null` check
  - Use `if` statement to check whether referent is `null`
  - If `null`, `then` clause is not executed

If *z* is not **`null`**, execute block.
This check is mandatory.

```
var z: Int? = 10
z = 20
if (z != null) {
    print("Z is " + z + ".")        // "Z is 20."
}
else {
    print("Bummer!")
}
```

29

2. Safe calls operator ?.

- Unary infix/postfix operator
  - Syntax: **?.**
  - Semantics: Return non-null value or **null**

```
val s2: String?
var s3: String? = null
s2 = "hello there!"

// Prints ”The length of s2 is 12."
println(”The length of s2 is " + s2?.length + ".")

// Prints ”The length of s3 is null."
println(”The length of s3 is " + s3?.length + ".")
```

Safe call operator returns 12.

Safe call operator returns `null`.

30

3. Elvis operator ?:

- Binary infix operator
- Implicitly cast nullable variable to non-null variable
  - Syntax: **?:** (binary infix operator)
  - Semantics: Return 2nd operand of **?:** if 1st operand is `null`; otherwise return 1st operand
  - A variation of the ternary operator?

```
val s4: String? = null

    // Prints ”The length of s4 is zero."
    println("The length of s4 is " + ((s4?.length) ?: "zero."))
```

Elvis operator turns `null` into `"zero."`.

Safe call operator may return `null`.

31

4. Bang bang operator !!

- Implicitly cast nullable variable to non-null variable
  - Syntax: `!!`
  - Semantics: Casts nullable variable to matching non-null type; throws `NullPointerException` if variable was null, otherwise return non-null value

32

4. Bang bang operator !! (cont'd)

- Examples of use of operator `!!`

```
val w: Int? = 100
// Prints "Constant w is 100."
println("Constant w is " + w!! + ".")
```

Bang bang operator – OK.

```
val w: Int?
// Forgotten w initialization
println("Constant w is " + w!! + ".")
```

Compile-time error.

```
val w: Int? = null
// Invalid cast
println("Constant w is " + w!! + ".")
```

Run-time NPE exception.

33

5. Safe casting (as? operator)

- *Casting* = Operation instructing compiler to change type of an identifier
  - Do not confuse *casting* with *conversions*: Casting does not create a new object at R-T, whereas a conversion does
  - Checked by Kotlin's run-time system; `ClassCastException` thrown if illegal cast specified (e.g., object is not target type of cast)
  - Code below would throw exception if **u** was **null**
  - Syntax: `<existing_identifer>` **as** `<new_data_type>`

```
val u: Int? = 100
val v: Int = u as Int
println("Constant v is " + v + ".")
```

Identifier *u* is cast to an *Int*.

34

5. Safe casting (cont'd)

- *Safe casting* = Similar to casting but return `null` instead of throwing exception if object is not target type of cast
  - Syntax: `<existing_identifer>` **as?** `<new_data_type>`
  - Semantics: Compiler will treat the existing identifier as an identifier of the new data type

```
val u: Int? = 100
val v: Int? = u as? Int
println("Constant v is " + v + ".")
```

Identifier *u* is cast to an *Int*.

35

Swift: ~~Nullable~~ Optional data types

- Swift uses optional data types similar to Kotlin's nullable types
- Wrapping references to optional identifiers is still required
- Wrapping can be avoided if programmer is certain that referent of optional-type identifier is not **`nil`**
  - Syntax: One Bang!
  - Programmer asserts that referent is not **`nil`** with the bang
  - Run-time error if variable was in fact **`nil`**
- Code below is Swift, not Kotlin

```
var z: Int? = 10
z = 20
z = z! + 15              // z is now 35
z = z + 15               // compiler error, no wrapping
```

36

Kotlin operators

1. Arithmetic
2. Relational
3. Logical

37

Arithmetic operators in Kotlin

- Syntax similar to Java, except for bitwise operators
  - + (binary), + (unary), - (binary), - (unary), *, /, %, ++ (prefix), ++ (postfix), -- (prefix), -- (postfix)
  - Combined with assignement: +=, -=, *=, /=, %=
- Bitwise operators: Binary infix functions applicable to `Int` and `Long` types
  - `shl` (Shift left signed)
  - `shr` (Shift right signed)
  - `ushr` (Shift right unsigned)
  - `and` (bitwise and)
  - `or` (bitwise or)
  - `xor` (bitwise exclusive or)
  - `inv` (bitwise negation)

38

Examples of bitwise operators

- Example of bitwise or – example of left shift

```
val u1: Int = 15                    // Int constant
val u2: Int = 19                    // Int constant
val u3: Int = u1 or u2              // Bitwise or
val u4: Int = u1 shl 2              // Left shift 2 bits

"Constant u3 is 31"
println("Constant u3 is " + u3 + ".")

"Constant u4 is 60"
println("Constant u4 is " + u4 + ".")
```

Can you guess *u3*'s value?

What about *u4*'s value?

39

Relational operators

- Syntax similar to Java, except for equality operators
  - Binary operators, >, >=, <, <=
  - Translated to `compareTo()` for reference types

Equality operators in Kotlin

- Two kinds of equality: *physical identity* and *logical equivalence*
- **Physical identity:** Two objects are equal if they are the same object
- **Logical equivalence:** Two objects are equal either if they are the same object, or if they are different objects with the same structure and content
  - This depends on the object type
  - E.g., 2 arrays equivalent if length, data type and values are the same
  - E.g., 2 person instances equivalent if same name, ID, DOB, etc.
- Primitive types (e.g., `Int`) are identical iff they are equivalent
  - Primitive types store directly their value; they are not references to objects

Equality operators in Kotlin (cont'd)

- Syntax
  - **Physical identity:** Operators === and !==
  - **Logical equivalence:** Operators == and !=
- See examples next...

42

Examples of equality operators

- Examples logical equivalences and physical identity

```
val u5: Double = 2.0
val u6: Double = 2.0
val u7: Double? = 2.0
val u8: Double? = 2.0
val u9: Double? = u7

u5 == u6      true
u5 === u6     true
u7 == u8      true
u7 === u8     false
u7 == u9      true
u7 === u9     true
```

Can you guess the value of these expressions?

43

Logical operators

- Traditional operators: `&&` (conjunction), `||` (disjunction), `!` (negation)
  - Lazy (short-circuit) conjuction and disjunction
- *Boolean* class also defines logical methods
- Binary infix operators (greedy, not lazy):
  - `and(Boolean) : Boolean` – Greedy conjunction
  - `or(Boolean) : Boolean` – Greedy disjunction
  - `xor(Boolean) : Boolean` – Exclusive or
- Unary negation `not()`
  - E.g., `true.not()` → `false`
  - Compare with binary infix syntax: `true and false` → `false`

44

Other operators

- Membership operators: `in, !in`
- https://kotlinlang.org/docs/reference/keyword-reference.html
- https://www.programiz.com/kotlin-programming/operators

45

Kotlin: Flow of Control

Flow of control: If statements

- Syntax: Same as Java
- Semantics: Choose branch as in Java + return a result (last block expression), similar to ternary
  - Must have “else” branch, if return result desired
  - Similar behavior to ternary operator “`?:`” in Java
  - No ternary operator in Kotlin; `if` expression does it all

```
min = if (a < b) { a }
      else { b }          // min assigned a value
```

If expression returns either *a* or *b*.

When statements

- Multi-branch conditional statements
- Two forms
  1. Variable controls choice of branch
     - Similar to Java's `switch` statement
  2. Guards (conditional expressions) control choice of branch

1. When as a switch statement (w/ variable)

- Syntax:

```
when (<variable>) {
     <value> -> <statement or block>
     <value> -> <statement or block>
     ...
else -> <statement or block> }
```

- Semantics: First branch whose value matches variable value is chosen, otherwise else clause chosen
- Returns value
  - In general, there should be an "else" branch
  - Can be used as an r-value

Examples of when statements

- Source https://kotlinlang.org/docs/reference/control-flow.html

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

50

Examples of when statements (cont'd)

- Range specs possible too
  - https://kotlinlang.org/docs/reference/control-flow.html

```
when (x) {
    in 1..10 -> print("x is in low half of range")
    !in 10..20 -> print("x is outside range")
    else -> print("none of the above")
}
```

51

Examples of when statements (cont'd)

- Type checks possible too
  - https://kotlinlang.org/docs/reference/control-flow.html
- Seems to duplicate message dispatching in OO languages

```
when (animal) {
   is Dog -> print("Woof, woof")
   is Cat -> print("Meow, meow")
   is Duck -> print("Quack, quack")
   else -> print("I keep quiet")
}
```

52

2. When as a multi-branch, guarded conditional

- Syntax:

```
when {
     <condition> -> <statement or block>
     <condition> -> <statement or block>
     ...
else -> <statement or block> }
```

- Semantics: First branch whose condition evaluates to true is chosen for execution
- This version also returns value

53

Example of when statement as guarded statement

- Guarded statements: A sequence of (condition, statement) pairs
  - Relational or logical expression controls branch to be executed
- Conditions evaluated in order in which they appear in guarded statement
  - First true condition executes corresponding statement
  - Similar to *cond* statement in LISP, or *if-elif-else* of Python

```
x = ...
when {
   x.isDigit() -> print("x is a number")
   x.isLetter() -> print("x is a letter")
   x.isWhitespace() -> print("x is white space")
   else -> print("x is none of the above")
}
```

54

Iteration statements

Three kinds of iteration

1. Indexed iteration (`for` loop with integer identifier and integer range)
2. Conditional iteration (`while` and `do-while` loops, with condition)
3. Iteration over collections (`for-in` loops, with *item* and collection of *item*)
   - Language supports `break` and `continue` statements

55

Indexed iteration

- Syntax: **`for`** loop similar as Java, except:
  - Index values specified with keyword **`in`** and range
  - Index type is inferred
- Semantics: Similar to Java
- Does not return value
  - Do not use as r-value

Type of *i* is *Int* (inferred).

```
var total = 0
for (i in 1..10) {
    total += i
}
```

total will be 55.

56

Conditional iteration: Initial condition loops

- Syntax: **`while`** loop works the same way as in Java

Type of *i* and *total* is *Int* (inferred).

```
var total = 0
var i = 1
while (i <= 10) {
    total += i++
    }
println("total is ${total}.")        // "total is 55."
```

Note expression evaluation embedded in string.

57

Conditional iteration: Final condition loops

- Syntax: **do-while** loop works the same way as in Java

Again inferred *Int* types.

```
var powerOfTwo = 1
var i = 0
do {
    powerOfTwo *= 2
        i++
}
while (powerOfTwo <= 100)

println( "powerOfTwo and i are $powerOfTwo and $i." )
                              // "powerOfTwo and i are 128 and 7."
```

What does this print?

58

Kotlin: Arrays

59

Array basics

- Indexed collections, stored contiguously
- Instances of template class **`Array<T>`**
  - Use this syntax to declare or define array identifiers
- Instance creation with function **`arrayOf(element1, element2, ...)`**
- Square bracket notation to access and modify array elements

```
var anArray = Array<Int>
anArray = arrayOf(1, 2, 3)
anArray[0] =   4
println(anArray[0])
```

Define identifier of `Int` array.

Create array instance with 3 elements.

Modify array element.

Access array element.

60

Array types

- Arrays are typed (all elements usually are the same type)
  - This is the norm in a statically typed language
  - Use type “**`Any`**” to mix types in arrays
  - **`Any`** means any data type (whether primitive or not)
- Also array identifiers are references, not values

```
var anotherArray: Array<Any>
anotherArray = arrayOf(3, 5.5, 'c', "hello there!”)
var arrayCopy: Array<Any>
arrayCopy = anotherArray
arrayCopy[2] = 'd’
println("anotherArray[2] = ${anotherArray[2]}.”)
```

Define first array identifier.

Define second identifier.

Assign second array.

Prints “anotherArray[2] = d”

61

More on array creation

- Create instance with given length, filled with **`null`**s with function **`arrayOfNulls(int)`**
- Class constructor **`Array(int, map)`** lets you define array of given length and initial values

Define identifier of Int array.

Create array instance with 10 elements.

```
var sevenMultiples: Array<Int>
sevenMultiples = Array(10, { i -> i * 7 } )
for (i in 0 .. 9)
    print("${sevenMultiples[i]} ")
```

Print array elements.

62

More on arrays

- Predefined types for arrays of primitive types
  - Avoid wrapper class overhead + simplify declarations
  - Use **`ByteArray, ShortArray, IntArray, DoubleArray`** instead of **`Array<Byte>, Array<Short>, Array<Int>, Array<Double>`**
- Buyers beware: The resulting arrays are not instances of **`Array<T>`** class
- Each array type has a factory function for instance creation
  - E.g., **`intArrayOf(3, 6, 9, 12),`** etc.

63

Functions, block closures and scope rules

Kotlin functions

- Four kinds of functions
  - File scope (not contained in any class or other function)
  - Member functions (i.e., methods in a class)
  - Inner functions (nested inside another function)
  - Extension functions (additions to an existing class)

Functions

- Kotlin peculiarities: Functions can be at file scope, class scope (e.g., methods) and nested in other functions (inner functions)
  - Java does not support file-scope and inner functions
- Syntax: functions are **fun** (pun intended)
  - Parameter types and return value declared as usual

```
fun main(args: Array<String>): Int {
    return min(10, 20) // function call
}

fun min(x: Int, y: Int): Int {
    if (x <= y) return x
    else return y
}
```

File scope functions.

Function header declares function name, parameter names and types, and return type (like Java).

66

Function parameters and return type

- Objective C-like syntax: unique keyword, a colon and a data type
  - Comma separated list
- Default values: equal sign = and default value
  - May omit argument in call if matching parameter has default value

```
fun distance(x1: Double, y1: Double,
             x2: Double = 0.0, y2: Double = 0.0): Double {
    val dx: Double = x1 - x2
    val dy: Double = y1 - y2
    return Math.sqrt(dx*dx + dy*dy)
}

// Example of call
distance(3.0, 4.0)
```

Default values for *x2* and *y2.*

Function call uses default values for arguments *x2* and *y2.*

67

Named vs. positional arguments

- Support for both positional arguments (the usual) and named arguments
  - Increase readability of function calls by naming parameters?
  - Syntax: Parameter keyword, equal sign, and argument value
  - Can mix, but then put positional arguments before named arguments

```
fun distance(x1: Double, y1: Double,
             x2: Double = 0.0, y2: Double = 0.0): Double {
    val dx: Double = x1 - x2
    val dy: Double = y1 - y2
    return Math.sqrt(dx*dx + dy*dy)
}

// Example of call
distance(x1 = 2.0, y1 = 1.5, x2 = -2.0, y2 = -1.5)
```

Default values for *x2* and *y2.*

Function call uses named arguments for all parameters.

68

`Unit`, aka the new `void`

- Function that does not return a value is said to return `Unit`
  - `Unit` return type can be omitted from function header
  - Could be called `void`, oh well…
- Reference: https://kotlinlang.org/docs/kotlin-docs.pdf

69

One-statement functions

- Omit braces, specify single statement after = sign
- Return type not specified explicitly, inferred from expression's return value
- Example: Euclidean distance between 2 points as one line

Double return type is inferred from expression.

```
fun distanceOneLine(x1: Double, y1: Double,
                    x2: Double, y2: Double)
    = Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
```

Use = sign and return expression instead of function body in curly braces.

70

Additional function modifiers

- **infix**: Function can be invoked with infix notation (like arithmetic shift operators)
- **tailrec**: Function compilation optimized for tail recursion
  - Tail recursive function: (1) One recursive call and (2) Call is last expression in the function

factorial is tail recursive.

```
tailrec fun factorial(n: Int) : Int {
    if (n <= 1) return 1
    else return n * factorial(n-1)
}
```

Last expression is the recursive call.

71

Functions as first-class objects

- First-class language object if it meets two conditions:
  1. Can be defined anywhere an object can be declared/defined
  2. Can be assigned to variables, passed to and from functions the way any other language object is
     - ➢ Treat a function object like you would for an *Int*, a *String*, etc.
- *High-order function*: A function that takes as parameter or returns another function
  - – High-order functions are possible in Kotlin
  - – A very powerful mechanism
- *Lambda expression*: Anonymous functions (1st-class objects, of course)

72

Defining variable bound to a function

- Syntax of type specification for function variable
  - – (*<par_type1>, <par_type2>, ...*) -> *<return_type>*
  - – Parameter names can be included, e.g.,

```
(x: Int, y: Int) -> Double
```

  - – Receiver type can be included, i.e., for class method

```
Point.(anotherPoint: Point) -> Double
```

  - – Specify nullable function type using parenthesis and ?

```
((x: Int, y: Int) -> Double)?
```

```
var myFun: ((Double, Double, Double, Double) -> Double)?
```

myFun will be bound to *Double* functions with 4 *Double* parameters.

myFun is nullable.

73

Binding function variable to function

- Several ways to do that
  - Create a lambda expression or anonymous function (will see later)
  - Use an existing function definition
- Syntax for existing function uses scope operator **::**
  - Syntax for for top-level or local function ::<*function_name*>
  - Syntax for class method: <*class_name*>::<*function_name*>

```
var myFun: ((Double, Double, Double, Double) -> Double)?

myFun = ::distance
var aDistance = myFun(0.0, 0.0, 3.0, 4.0)
```

Assigning a function to a function variable.

Calling a function through the function variable.

74

Calling function through function variable

- Two ways
  - Use *invoke()* operator on variable
  - Call it directly through variable (as we saw before)
- Example using *invoke()*

```
var myFun: ((Double, Double, Double, Double) -> Double)?

myFun = ::distance
var aDistance = myFun.invoke(0.0, 0.0, 3.0, 4.0)
```

Function assignment.

Explicit use of *invoke()* operator.

75

Lambdas

- Function literals specifying executable statements
  - Can take parameters, return a value
  - Free variables in lambdas use static scope rules (references resolved in context where lambda is defined, not where lambda is called)
  - Again, first-class Kotlin objects
- Syntax: `{` *<par_list>* `->` *<return_type>* `=` *<expressions>* }
  - *par_list* is comma-separated list of *par_name*: *par_type*
  - *expressions* are as usual
  - Last expression value is returned
- Example 1: `{ x: Double, y: Double -> x * y }`
- Example 2: `{ x, y-> free_variable = 100; x * y }`

76

Parameter *it*

- Lambda with one input parameter may omit parameter declaration and -> syntax
  - Use "it" as parameter name
  - Works only if compiler can infer *it*'s type from context
  - Again, lambdas are first-class Kotlin objects

Compiler infers *Double* type of *it* from context.

```
var myFun3: (Double) -> Double
myFun3 = {it * it * it}
println(myFun3(5.0))
```

Prints 125.0.

77

Special syntax of lambda arguments

- Function taking a single lambda expression as parameter can be called with different syntax placing lambda expression out of argument list
- Iterator *forEach()* applies lambda expression to every element of an array
  - Use "it" as parameter again
  - Place argument lambda outside argument list (omitted altogether)

```
var anArray: Array<Double> = arrayOf(10.0, 20.0, 30.0)
var total = 0.0
anArray.forEach {total = total + it}
```

Argument list of *forEach* call is omitted.

Argument lambda specified after *forEach*'s identifier.

78

Anonymous functions

- Functions without a name
  - Similar to lambdas
  - Can specify return type (which lambdas cannot)
  - Syntax: usual function syntax, just omit name

```
var myFun4: (Double) -> Double ;

// Anonymous function has no name
myFun4 = fun (x: Double)  =  (x * x * x)

println("myFun4 returns ${myFun4.invoke(5.0)}.")
```

79

Kotlin	Swift	Objective-C
class	class	@interface
interface	protocol	@protocol
constructor/create	Initializer	Initializer
Property	Property	Property
Method	Method	Method
@Throws	throws	error:(NSError**)error
Extension	Extension	Category member
companion member <-	Class method or property	Class method or property
null	nil	nil
Singleton	Singleton()	[Singleton singleton]
Primitive type	Primitive type / NSNumber	
Unit return type	Void	void
String	String	NSString
String	NSMutableString	NSMutableString
List	Array	NSArray
MutableList	NSMutableArray	NSMutableArray
Set	Set	NSSet
MutableSet	NSMutableSet	NSMutableSet
Map	Dictionary	NSDictionary
MutableMap	NSMutableDictionary	NSMutableDictionary
Function type	Function type	Block pointer type

Mapping Kotlin to Swift and Objective C

https://kotlinlang.org/docs/kotlin-docs.pdf

80

Classes and Inheritance

81

Kotlin's object model

- Single inheritance model + abstract interfaces
  - Class can inherit from just one superclass + zero, one or many interfaces
- By default classes cannot be subclassed
  - Declare a superclass to be `open` if it should be used as a superclass
- By default methods cannot be refined/overridden in a subclass
  - Use `open` specifier before superclass's method, `override` specifier before subclass's method
- Primary constructor vs. secondary constructors
  - Secondary constructors must invoke primary
- Init block is executed before constructors

82

Class example

- Fields (called properties) can be constant (**val**) or mutable (**var**)
  - Similar to lambdas

A constant property *x*

```
class C {

   val x : Int

   init { println("Init block") }

   constructor(i: Int) {
       println("Constructor")
       this.i = i  }
}
```

Init block is executed before constructors

Assign second array.

Prints "anotherArray[2] = d"

83

And this concludes our program…

Thank you very much!