Kotlin

amit.gulati@gmail.com

1

Kotlin

- What is Kotlin?
  - Programming Language by Jetbrains
  - 1.0 Officially announced in 2016
  - Open Source
  - Inspired by Java, Scala, Groovy, C# etc.
  - Compiles to JVM compatible bytecode
  - Fully InterOp with Java

2

Kotlin

- Goals
  - Concise
    - No Semi-Colons
    - Much less boiler plate code
    - Syntax sugar
  - Safer alternative to Java
    - Support for nullable types
  - Modern Programming Language
    - Lambda, Closures, Functional programming support etc.

3

3

Kotlin

- Documentation
  - www.kotlinang.org

4

4

Kotlin

- Kotlin File
  - .kt
  - Global named values, function definitions, class definitions, interface definitions

```
var myGlobalVariable:Int = 99

fun main() {
}

fun myFunction() {
}

class MyClass {
}

interface MyInterface {}
```


6

Kotlin

- Entry Point of a Kotlin program
  - main function

```
fun main() {
    print("Hello World")
}
```

  - One file can have only one main function

7

Kotlin

- Coding Convention
  - Semi-colons are optional
  - Kotlin follows Java naming convention
    - Camel Case
    - Types begin with upper case
    - Variables and function names begin with lowercase
    - Packages follow the reverse domain name notation

8

Comments

- C style comments

```
//single line comment

/*
    Multiline comment
 */

/*
    Multiline comment
    /*
        Nested Multiline comment
     */
 */
```


10

Values and Expressions

- What does the following program do?

```
println(3.14159 * 25.0 * 25.0)
println(2 * 3.14159 * 25.0)
```

  - Magic values in programs
    - Program is difficult to understand
    - Error prone
  - Replace magic values with named values

```
val PI = 3.14159
var radius = 25.0
val area = PI * radius * radius
val perimeter = 2 * PI * radius
println(area)
println(perimeter)
```

11

11

Named Values

- **var** keyword
  - Declare mutable type or variable

```
var name = "kotlin"
```

  - Can be reassigned

```
name = "kotlin 1.2"
```

  - Cannot declare more than one variable names in a single line

```
var name, version
```

Unexpected tokens (use; to separate expressions on the same line

12

12

Named Values

- **val** keyword
  - Declaring immutable type or values

```
val pi = 3.141
```

  - Can only be assigned once.

```
pi = 3.141
```

val cannot be reassigned

  - Only makes the variable or reference a constant, not the object referenced

```
val message = StringBuilder("Hello ")
//message = StringBuilder("another")
message.append("World")
```

13

13

Named Values

- Naming Convention
  - Can contain almost any character, including Unicode characters
  - Cannot contain whitespace characters, mathematical symbols, arrows, etc.

14

14

Type System

- Type Representation
  - In Kotlin everything is an object
    - No special primitive types
    - Internally some types may be represented as primitives on the JVM
  - Any is the (implicit) base class
  - Compiler does not do implicit type expansion

15

Type System

- Numeric Types

Type	Size
Byte	8
Short	16
Int	32
Long	64
Float	32
Double	64

```
123
123456L
0xAB
0b01010101

12.34
12.34F
123.5e10
```


16

Type System

- Boolean Type

```
true
false
```

  - Result of logical expressions

```
x == y , x < y, x > y
```

  - Conjunction (&&), and disjunction (!!) operations

```
x < y && x < z
x == y || y == z
```

17

17

Type System

- Character Type
  - Character literals use single quotes

```
'A', 'B', 'C', 'D', 'E'
```

- String Type
  - Ordered Collection of Characters enclosed in double quotes

```
"Hello, world!\n"
```

18

18

Type System

- Type System
  - Kotlin is a strongly typed language
  - Named values must have a type
    - Type cannot change at runtime
  - Type for named values can be provided in 2 ways
    - Type Inference
    - Type Annotation (Explicit Type Definition)

19

19

Named Values with Types

- Type Inference
  - Kotlin compiler is able to infer the type of a variable

```
val greet = "hello"

println(greet)
println(greet::class)
println(greet.javaClass)
```

  - Value must be assigned for the compiler to infer the type of variable

```
var name
```

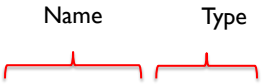
This variable must have a type annotation or must be initialized

20

20

Type System

- Explicit Type Annotation
  - Define the type for a named value




```
var isVisible:Boolean

var velocity:Float
var age:Int
var name:String
```

21

21

Named Values

- Warnings

```
fun main(args: Array<String>) {
    var myName = "Amit Gulati"
}
```

⚠ Parameter 'args' is never used :1
⚠ Variable 'myName' is never used :10

```
fun main() {
    val myName = "Amit Gulati"
    println(myName)
}
```

22

22

Type Safety

- Compile time overflow detection
  - Kotlin will detect overflow error when assigning values

```
var count:Byte = 300
```

This integer literal does not conform to the expected type Byte

```
var count:Short=99999
```

This integer literal does not conform to the expected type Short

23

23

Type Safety

- Implicit Type Conversion
  - Kotlin does not support implicit type conversion

```
var myInt:Int = 10
var myLong:Long = myInt
```

Type mismatch

  - Kotlin requires explicit type casting
    - Every variable type contains methods to convert it to other types

```
toLong()    toShort()    toChar()
toInt()     toFloat()
toByte()    toDouble()
```

24

24

Type Equality

- Two ways of Equality
  - == operator (Structural Equality)

```
val number1 = 100.6
val number2 = 100.6
println(number1 == number2)

true


val string1 = StringBuffer("Kotlin").toString()
val string2 = StringBuffer("Kotlin").toString()
println(string1 == string2)

true
```

25

25

Type Equality

- Two ways of Equality
  - === operator (Referential Equality)
    - Compares references

```
val string1 = StringBuffer("Kotlin").toString()
val string2 = string1
println(string1 === string2)

true


val string1 = StringBuffer("Kotlin").toString()
val string2 = StringBuffer("Kotlin").toString()
println(string1 === string2)

false
```

26

26

String formatting

- String Templates
  - Create String value from a mix of constants, variables, literals, and expressions
  - $ symbol is used to create a Template expression

```
val s = "abc"
val str = "$s.length is ${s.length}"
```

```
val side = 100
print("Area of Square with side = $side is ${side * side}")
```

27

27

Array Type

- Array
  - Represented by **Array** class (kotlin.Array)
  - Ordered collection
  - Creating
    - Library function **arrayOf**() to create an array of values

```
val numbers = arrayOf("One", "Two", "Three")
```

    - Type of the above array is `Array<String>`

28

28

Array Type

- Array

```
var arr = arrayOf(1, 2, 3, 4)
```

  - Array of boxed Integer types

Kotlin Code

```
fun sum(numbers:Array<Int>):Int {
    var result = 0
    for (number in numbers) {
        result += number
    }
    return result
}
```

Java Byte Code

```
int result = 0;
Integer[] var4 = numbers;
int var5 = numbers.length;

for(int var3 = 0; var3 < var5; ++var3)
{
   int number = var4[var3];
   result += number;
}

return result;
```

29

29

Array Type

- Array
  - Arrays of primitive types so that boxing-unboxing can be avoided.
    - IntArray
    - ShortArray
    - ByteArray
  - Creating array of primate types
    - intArrayOf `intArrayOf(1, 2, 3, 4)`
    - shortArrayOf `shortArrayOf(1, 2, 3, 4)`
    - byteArrayOf `byteArrayOf(1, 2, 3, 4)`

30

30

Nullable Types

- What are Nullable Types?
  - Help Eliminate Null Pointer Exceptions
  - Not every variable in Kotlin can be assigned **null**

```
var a: String = "abc"
a = null
```

Null cannot be a value for a Non-Null type String

  - Nullable References are marked using ?

```
var b: String? = "abc"
b = null
```

  - Any type can be marked as a nullable

31

31

Nullable Types

- Accessing Nullable Type
  - Without null check

```
var rect:Rectangle1? = null
rect.area()
```

Only safe calls or non-null asserted calls allowed on nullable

32

32

Nullable Types

- Accessing Nullable Type
  - With null check

```
var b: String? = "abc"
b = null
val len = if (b != null) b.length else -1
```

  - Safe calls using ?.

```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length)
```

33

33

Nullable Types

- Performing Operations on Nullable Type
  - Perform Operation on non-null value

```
val listWithNulls: List<String?> =
                                listOf("Kotlin", null)
for (item in listWithNulls) {
    item?.let { println(it) }
}
```

35

35

Nullable Types

- Elvis Operator and Nullable types
  - If not null use value else another value

```
val l: Int = if (b != null) b.length else -1
```

  - Using elvis operator ?:

```
val b: String? = null
val l = b?.length ?: -1
```

36

36

Nullable Types

- Not-Null Assertion Operator (!!)
  - Converts any value to a non-null type and throws an exception if the value is null

37

37

Package

- Default package
  - No **package** directive provided in a file

```
fun myFunc() {
    print("Hello World")
}
```

src/main/kotlin/myfile.kt

  - Symbols added to default namespace

Package

- **package** directive
  - Source file may start with a package directive

src/main/kotlin/myfile.kt

```
package com.example.mypackage

fun myPackageFunc() {
    print("Hello World")
}
```

Package directive does not match file location

  - All contents of the file will belong to that package

```
com.example.mypackage
```

Package

- **import** directive
  - Each file may contain its own import directives
  - Importing a single symbol

```
import com.testing.mypackage.myPackageFunc
```

  - Importing more than one symbols

```
import com.testing.mypackage.*
```


41

Operators

Amit.gulati@gmail.com

42

Operators

- Arithmetic Operators

Operator	Meaning
+	Addition (also used for string concatenation)
-	Subtraction Operator
*	Multiplication Operator
/	Division Operator
%	Modulus Operator

43

43

Operators

- Compound Assignment Operators

Expression	Equivalent to
a +=b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

44

44

Operators

- Comparison and Equality Operators

Operator	Meaning	Expression
>	greater than	a > b
<	less than	a < b
>=	greater than or equals to	a >= b
<=	less than or equals to	a < = b
==	is equal to	a == b
!=	not equal to	a != b

45

45

Operators

- Logical Operators

Operator	Description	Expression
||	true if either of the Boolean expression is true	(a>b)||(a<c)
&&	true if all Boolean expressions are true	(a>b)&&(a<c)

46

46

Operators

- Range Operators
  - Defines a range that runs from a to b, and includes the values a and b.
  - Integral type ranges are represented by classed **IntRange**, **LongRange**, **CharRange**

```
var range = 0..10

var charRange = 'a'..'z'

var longRange = 0L..100L
```

47

47

Expressions and Statements

- Expression vs Statements
  - Statements and expressions are the smallest useful fragments of code in most programming languages.
- Statement
  - A statement has an effect, but produces no result.
  - A statement is always a top-level element.
- Expression
  - An expression produces a value, which can be assigned or used as part of another

48

48

Control Flow

amit.gulati@gmail.com

49

Control Flow

- Branching
  - if
  - when
- Loops
  - for
  - While
- Transfer Flow
  - continue
  - break

50

Control Flow

- **if** expression

```
if (condition) {
}
```

  - The conditional expression inside the parentheses after the if must evaluate to true or false.
  - The else keyword allows you to handle both true and false paths.

```
if (condition) {
} else {
}
```


Control Flow

- **if** expression
  - Curly brace is not mandatory.

```
val n: Int = -11
if (n > 0)
    println("It's positive")
else
    println("It's negative or zero")
```

Control Flow

- **if** expression
  - Unlike Traditional Usage in which **if** is a statement

```
var max = a
if (a < b) max = b
```

  - **if** is an expression in Kotlin
    - Expression evaluates to a value

```
val max = if (a > b) a else b
```

53

53

Control Flow

- **if** expression
  - Evaluates to a value

Single line if expression

```
var a = 10
var b = 20
val max = if (a > b) a else b
```

When assigning result of if expression to a variable, else is a must

Block if expression

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

54

54

Control Flow

- **if** expression
  - No need for a ternary operator

```
var str = "Hello World"
var result = if (str.length < 10) true else false
```

55

55

Control Flow

- **if-else** expression

```
fun main() {
    val n: Int = -11
    if (n > 0)
        println("It's positive")
    else if (n == 0)
        println("It's zero")
    else
        println("It's negative") }
```

56

56

Control Flow

- **when** expression
  - Replaces the switch statement of C-like languages

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

  - Can be used either as an expression or as a statement.
  - **else** branch is evaluated if none of the other branch conditions are satisfied

57

57

Control Flow

- **when** expression
  - Multiple matches

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

  - Expressions as matching values

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

58

58

Control Flow

- **when** expression
  - Range as matching values

```
when(temperature) {
        in Float.MIN_VALUE..60.0f -> "Too Cold"
        in 70.0f..Float.MAX_VALUE -> "Too Hot"
        in 60.0f..70.0f -> "Just Right"
        else -> "Not Sure"
```

59

59

Control Flow

- **for-in**

```
for (index in range)
```

  - Forward Iteration

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

  - Backward Iteration

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

60

60

Control Flow

- **for-in**
  - Skipping Values

```
for (i in 1 until 10) {
       println(i)
}

for (i in 1 until 10 step 2) {
    println(i)
}
```


61

Control Flow

- **for-in**

```
for (i in array.indices) {
       println(array[i])
}

for ((index, value) in array.withIndex()) {
       println("the element at $index is $value")
}
```


62

Control Flow

- **while** loop
  - Evaluates its condition at the start of each pass through the loop.

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

  - **do-while** evaluates its condition at the end of each pass through the loop.

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

63

63

Control Flow

- **continue** statement
  - Stop the current iteration and move to next iteration

```
fun count(name:String, names:Array<String>):Int {
    var counter = 0
    for (n in names) {
        if (n == name)  {
            counter++
            continue
        }
    }
    return counter
}
```

64

64

Control Flow

- **break** statement
  - Terminates the execution of an entire control flow statement.
  - **break** statement inside a loop will terminate the loop

```
fun nameExists(name:String, names:Array<String>):Boolean {
    var exists = false
    for (n in names) {
        if (n == name)  {
            exists = true
            break
        }
    }
    return exists
}
```

65

65

Control Flow

- Control Transfer Statements and Labels
  - Any expression in Kotlin may be marked with a label.
  - Labels is identifier followed by the @

```
loop@ for (i in 1..100) {
    for (j in 1..100) {
        if (j % i == 0 ) break@loop
    }
}
```

66

66

Functions

amit.gulati@gmail.com

67

Functions

- Defining Functions

```
fun multiply (x: Int, y: Int) : Int
{
}
```

  - Use **fun** keyword to declare a function
  - Function has a **name**
  - Optional one or more named, typed input parameters.
  - Optional typed return value.
  - Optional curly braces that contain the function body

68

68

Functions

- Defining Functions
  - Function not taking any parameter and not returning anything

```
fun printHello(): Unit {
        println("Hello!!")
}
```

  - Unit can be omitted from function signature

```
fun printHello()
```

  - Calling the function

```
printHello()
```

69

69

Functions

- Defining Functions
  - Function taking parameter and returning a value

```
fun greet(name:String):String {
    return "Hello !! $name"
}
```

    - Parameters are defined using Pascal notation, i.e. *name*: *type*
  - Calling the function

```
val message = greet("John")
print(message)
```

70

70

Functions

- Single Expression Functions
  - Function returns a single expression

```
fun greet(name:String):String {
    return "Hello !! $name"
}
```

  - Replace it with a Function Expression

```
fun greet(name:String):String = "Hello !! $name"
```

  - Return type is inferred by compiler

```
fun greet(name:String) = "Hello !! $name"
```

71

71

Functions

- Default Arguments
  - Function parameters can have default values.
  - Specified using the assignment operator

```
fun greet(name:String, greeting:String="Hello")
```

  - Used when a corresponding argument is omitted.

```
greet("Amit")

name = "Amit"
greeting = "Hello"
```

72

72

Functions

- Default Arguments
  - Parameters with default values are placed towards the end of the parameter list.
  - If placed in the beginning of parameter list

```
fun greet(greeting:String="Hello",name:String)

 greet("Amit")
```

No value passed for parameter name

73

Functions

- Named Arguments
  - Function definition with multiple parameters

```
fun createPerson(name:String, age:Int, height:Int, weight:Int){
    println("$name $age $height $weight")
}
```

  - Calling the above function

```
createPerson("John", 20, 163, 75)
```

  - Calling with named arguments

```
createPerson(name = "John", age = 20, height = 163, weight = 75)
```

74

Functions

- Named Arguments
  - Calling with named arguments
    - All the positional arguments should be placed before the first named one

```
f(1, y = 2) is allowed, but f(x = 1, 2) is not
```

    - Named argument syntax cannot be used when calling Java functions

Functions

- Variable Arguments
  - Provide a comma separated list of arguments
  - Mark the parameter using the keyword **vararg**

```
fun printNames(vararg names:String) {
    for (name in names) {
        println(name)
    }
}
```

  - Inside the function a vararg-parameter of type T is visible as an array of T

Functions

- Destructuring

```
fun minMax(numbers:IntArray):Pair<Int, Int>
```

  - Without Destructuring

```
val result = minMax(intArrayOf(100, 34, 99, 20, 5))
println("${result.first}, ${result.second}")
```

  - With Destructuring

```
val (min, max) = minMax(intArrayOf(100, 34, 99, 20, 5))
println("$min, $max")
```

77

77

Tuple

79

79

Tuple

- Tuple
  - Tuples are sequences of objects of small, finite size.
  - Kotlin provides two specific types:
    - Pair for a tuple of size two
    - Triple for a size of three.

80

Tuppe

- Pair
  - Pair 2 values of same or different types
    - Pair<A, B>

```
Pair<Int, Int>(100, 99)
```

    - Triple<A, B, C>

```
Triple<Int, String, Char>(1, "One", '0')
```

  - Function returning multiple values

```
fun minMax(numbers:IntArray):Pair<Int, Int>
```


81

Collections

amit.gulati@gmail.com

82

Collections

- Java Collections
  - Java collections are available in Kotlin (ArrayList, Map, Set etc.)
  - Java provides a combined (mutable/immutable) interface for collection types
    - Makes it difficult to trap unexpected operations at compile time
    - If we call an add method on a immutable collection, UnsupportedOperationException is thrown.

83

Collections

- Kotlin Collections
  - Kotlin provides additional convenience methods to what Java provides
  - Kotlin provides both Mutable and Immutable flavor for Collection Interfaces
  - Kotlin collection classes internally map to Java collection classes
  - Collections in Kotlin (kotlin.collection package)
    - **List** - ordered collection of objects.
    - **Set** - unordered collection of objects.
    - **Map** - associative dictionary or map of keys and values.

Collections

- List
  - Ordered collection
  - Mutable or Immutable

    MutableList, **List**

    both are Kotlin views around ***ArrayList***
  - Convenience methods to create list or a mutable list
    - **listOf**
    - **mutableListOf**

```
val names = listOf("Raj", "Joe", "John")

val names = mutableListOf("Raj", "Joe", "John")
```

Collections

- Map
  - Collection of Key-Value pair
  - Mutable or Immutable
  - Creating a Map
    - mapOf
    - mutableMapOf

```
val airports = mapOf("DEL" to "New Delhi",
                     "BOM" to "Mumbai")

val airports = mutableMapOf("DEL" to "New Delhi",
                            "BOM" to "Mumbai")
```

86

86

Collections

- Set
  - Unordered collection of elements that does not support duplicate elements.
  - Mutable or Immutable
  - Creating a Set
    - setOf
    - mutableSetOf

```
var numbers = setOf("One", "Two", "Three", "One")
println(numbers.toString())
[One, Two, Three]
```

```
var numbers = mutableSetOf("One", "Two", "Three", "One")
println(numbers.toString())
[One, Two, Three]
```

87

87

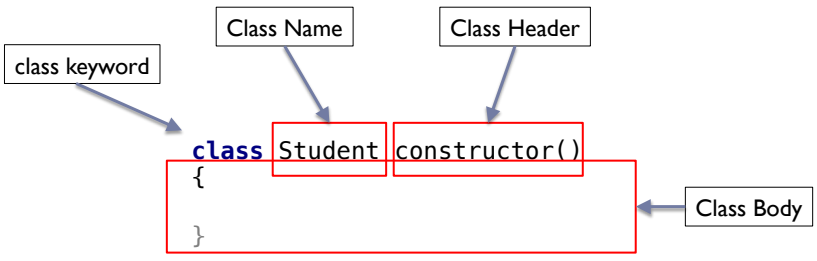
Classes and Objects

amit.gulati@gmail.com

88

Classes and Objects

- Class Definition



- Only thing that is mandatory is the class keyword and class name

89

89

Classes and Objects

- Class Definition

```
class Student {
}
```

  - Added to default package
  - In-fact the curly braces are not required

```
class Student
```

90

90

Classes and Objects

- Class Definition & files
  - Extension of a kotlin file is "**kt**"
  - Define multiple classes in the same file

University.kt

```
class Student {
}
class Teacher {
}
```

91

91

Classes and Objects

- Class with Properties
  - Define the state / attributes of the class
  - Can be mutable (**var**) / immutable (**val**)

```
class Student {
    var firstName:String
    var lastName:String
}
```

Property must be initialized or be abstract

92

92

Classes and Objects

- Class with Properties
  - Properties must be initialized
    - As part of declaration

```
class Student {
    var firstName:String = ""
    var lastName:String = ""
}
```

    - Constructor / initializer (more on this later)

93

93

Classes and Objects

- Class with Properties and Methods
  - Methods are functions that are part of class definition
  - Define the behaviors of a class

```
class Student {
    var firstName:String = ""
    var lastName:String = ""
    fun printFullName() {
        println("$firstName $lastName")
    }
}
```


Classes and Objects

- Default Constructor
  - Default constructor is synthesized for
    - Non-abstract class that does not declare any constructor (primary or secondary)
    - All properties have an initial value

```
class Student {
    var firstName: String = ""
    var lastName: String = ""
}
```

Classes and Objects

- Primary Constructor
  - Declared as part of class header
  - One per class

```
class Student constructor() {
}
```

  - **constructor** keyword is optional.

```
class Student() {
}
```


96

Classes and Objects

- Primary Constructor
  - A non-abstract class will have a generated primary default constructor with no arguments.

```
class Student {
}
```


97

Classes and Objects

- Primary Constructor with parameters

```
class Student(firstName:String, lastName:String) {
    var firstName:String = firstName
    var lastName:String = lastName

    fun printFullName() {
        println("$firstName $lastName")
    }
}
```


98

Classes and Objects

- Initializer block

```
class Student (firstName:String, lastName:String) {
    var firstName:String = firstName
    var lastName:String = lastName
    var fullName:String
    init {
        fullName = "$firstName $lastName"
    }
```

- Code in initializer blocks becomes part of the primary constructor
- Has access to parameters of primary constructors

99

Classes and Objects

- Initializer block

```
class InitOrderDemo(name: String) {

    val firstProperty = "First property: $name”

    init {
        println("First initializer block ${name}")
    }

    val secondProperty = "Second property: ${name.length}"

    init {
        println("Second initializer block ${name.length}")
    }
}
```


Classes and Objects

- Primary Constructor with Properties

```
class Student (var firstName:String, var lastName:String)
{
    var fullName:String

    init {
        fullName = "$firstName $lastName"
    }
}
```

  - Concise form for declaring properties and initializing them using primary constructor
  - Properties can be marked with **val** or **var**

Classes and Objects

- Primary Constructor and visibility
  - Changing the Visibility of primary constructor

```
class DontCreateMe private constructor () { }
```

102

Classes and Objects

- Secondary Constructor

```
class Student(var firstName:String, var lastName:String) {
    var fullName:String
    init {
        fullName = "$firstName $lastName"
    }

    constructor(): this("", "") {
    }
}
```

  - Created with keyword **constructor**
  - Delegate to primary constructor using **this** keyword

103

Classes and Objects

- Creating Instance
  - Call the constructor as if it were a regular function

```
var student = Student("John", "Doe")
```

    - Note that Kotlin does not have a **new** keyword.
  - Call constructor using named arguments

```
val student = Student(firstName = "Amit",
                     lastName = "Gulati")
```

104

Classes and Objects

- Referring to Properties
  - **dot** operator

```
var address = Address()
print("${address.name}")
```

105

Classes and Objects

- Properties in Java
  - Fields are used for defining state.
  - Getter/Setter are added to provide access to the field
  - Combination of Field + getter/setter is referred as property

```
public class Rectangle {
    private int width;
    private int height;

    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public void setHeight(int h) { height = h; }
    public void setWidth(int w) { width = w; }
}
```

106

106

Classes and Objects

- Properties in Kotlin
  - Declared using var or val keywords
  - Backing field, getter and setter generated in byte code.

```
class RectangleKt {
    var width = 100
    var height = 100
}
```

```
public final class RectangleKt {
        private I width
        public final getWidth()I
        public final setWidth(I)V
        private I height
        public final getHeight()I
        public final setHeight(I)V
```

107

107

Classes and Objects

- Property Getter / Setter

```
class Rectangle {
    var width = 100
        get() { return field }
        set(value) { field = value }

    var height = 100
        get() = field
        set(value) { field = value }
}
```

  - Backing store is referred to as **field**
  - **field** identifier can only be used in the accessors of the property.

108

108

Classes and Objects

- Property Getter / Setter
  - Implementing the getter/setter without using field

```
class Temperature(var celsius:Float) {
    var fahrenheit:Float
        get() = (celsius * 1.8f) + 32.0f
        set(value) {celsius = (value - 32.0f) / 1.8f}
}
```

  - In this case backing store is not created

109

109

Classes and Objects

- Property Getter / Setter
  - Readonly Properties

```
class Rectangle {
    var width = 100
    var height:Int = 100

    val area:Int
        get() = width * height
}
```


Classes and Objects

- Property getter/setter
  - Changing the visibility of getter/setter for a property

```
class Rectangle {
    var width = 100
    var height:Int = 100

    val area:Int
        get() = width * height

    var name = "Rectangle"
        private set
}
```

Classes and Objects

- Late Initialization of properties
  - Declare a non-null property and not initialize it

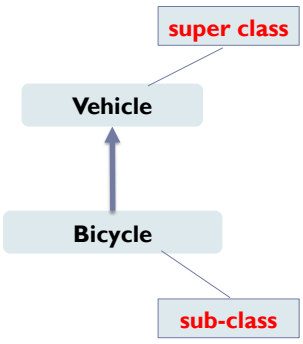
```
public class MyTest {
    lateinit var subject: String
}
```

  - Requirements
    - var properties declared inside the body of a class (not in primary constructor)
    - Non-null
    - Not primitive type

Classes and Objects

- Inheritance
  - Is-A relationship between classes
  - Child class inherits properties and methods of the parent.

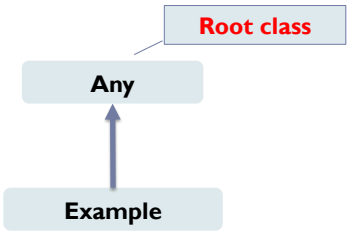
```
open class Vehicle {
    var currentSpeed = 0.0

    fun makeNoise() {
    }
}

class Bicycle : Vehicle() {

}
```



Classes and Objects

- Inheritance
  - Root class
    - Kotlin classes have a common super class i.e. **Any**

```
class Example {
}
```



    - Any is not java.lang.Object

117

117

Classes and Objects

- Inheritance
  - By default, all classes in Kotlin are final
  - Explicitly open class for inheritance using the **open** keyword

```
open class Base(p: Int)
```

  - Once a class is open for inheritance, then we can create sub-classes

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

118

118

Classes and Objects

- Inheritance and Initialization
  - Derived class must initialize the base class by calling the primary constructor

```
open class View(var context:Context) {
}

class MyView (context: Context): View(context) {
}
```


Classes and Objects

- Allow override of methods
  - Explicit annotation (**open** keyword) for overridable member functions

```
open class Base {
    open fun v() { }
    fun nv() { }
}
```

Classes and Objects

- Overriding methods
  - Explicit annotation (**override** keyword) for overriding an open method in super class

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

121

121

Classes and Objects

- **super** keyword
  - Call superclass functions and properties

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}
class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }
}
```

126

126

Classes and Objects

- Interfaces
  - Declarations of abstract methods and properties

```
interface IDrawable {
    var isVisible:Boolean
    fun draw()
}
```

  - Implementation of methods

```
interface IDrawable {
    var isVisible:Boolean
    fun draw()
    fun preDraw() { println("IDrawable:preDraw")
    }
}
```


Classes and Objects

- Properties in Interfaces
  - Abstract, or provide implementations for accessors.
  - Can't have backing fields

```
interface MyInterface {
    val prop: Int // abstract
    val propertyWithImplementation: String
        get() = "foo"
}

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

Classes and Objects

- Implementing Interfaces

```
class Rectangle : IDrawable {
    override var isVisible:Boolean = false
    override fun draw() { println("Rectangle:draw()") }
}
```

  - Methods implemented in the interface are also inherited

```
val rect = Rectangle()
rect.draw()
rect.preDraw()
```


Classes and Objects

- Resolving Overriding Conflicts

```
interface A {
    fun foo() { print("A") }

}
```

```
interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}
```

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

  - D has to override method named foo()

Classes and Objects

- Object Expression
  - Group a few local variables together as a logical entity.

```
val rectangle = object {
    var originX: Int = 0
    var originY: Int = 0
    var width: Int = 10
    val height: Int = 20
}
```

  - There is no type information.
  - Not very useful because of limitations

135

135

Classes and Objects

- Object Expression
  - Object Expression implementing an interface

```
val runnable = object: Runnable {
    override fun run() {
        println("Runnable executing")
    }
}
```

  - Type information attached to object
    - Passed to functions
    - Returned from functions

```
val thread = Thread(runnable)
thread.start()
```

136

136

Classes and Objects

- Object Expression
  - Object Expression implementing multiple interfaces

```
var runnable:Runnable = object : Runnable, AutoCloseable {
    override fun run() {
        println("Runnable executing")
    }
    override fun close() {
        println("Closing")
    }
}
```


Classes and Objects

- Object Expression
  - Object Expression implementing interface
    - SAM (Single Abstract Method) interface

```
interface Runnable {
    fun run()
}


val runnable = Runnable {
    println("Runnable executing")
}
```

Classes and Objects

- Object Expression
  - Object Expression can extend a class

```
var thread = object : Thread() {
    override fun run() {
        println("Thread with name = $name running")
    }
}
```


Classes and Objects

- Object Expression
  - use object expressions anywhere you'd use anonymous inner classes in Java.

Classes and Objects

- Object Declaration
  - **object** keyword followed by name

```
object Counter {

}
```

  - Properties and methods

```
object Counter {
    val counter = AtomicInteger()
    fun increment() {
        counter.getAndIncrement()
    }
}
```

141

141

Classes and Objects

- Object Declaration
  - init block

```
object Counter {
    val counter:AtomicInteger

    init {
        counter = AtomicInteger()
    }

    fun increment() {
        counter.getAndIncrement()
    }
}
```

142

142

Classes and Objects

- Object Declaration
  - Accessing object declaration properties

```
Counter.counter
```

  - Calling object declaration methods

```
Counter.increment()
```

Singleton pattern in Kotlin can be implemented using object declaration

Classes and Objects

- Object Declaration
  - Objects can be defined as a sub-type

```
open class Shape {
    var originX:Int = 0
    var originY:Int = 0
}
```

```
object Rectangle : Shape() {
    var width:Int = 0
    var height:Int = 0
}
```

  - Objects can implement Interfaces

```
object DoSomething : Runnable {
    override fun run() { println("Working") }
}
```

Classes and Objects

- Companion object
  - Declared inside class using **companion** keyword

```
class Student private constructor(val name: String) {
    companion object {
        fun create(name: String): Student {
            return Student(name)
        }
    }
}
```

145

145

Classes and Object

- Companion Object
  - Properties and Methods defined in companion object can be accessed using the class name.

```
Student.create("John Doe")
```

146

146

Classes and Object

- Companion Object
  - Named Companion object

```
class Student private constructor(val name: String) {
    companion object StringFactory {
        fun create(name: String): Student {
            return Student(name)
        }
    }
}
```

  - Using a named companion object

```
Student.StudentFactory.create("John Doe")
```

147

147

Classes and Objects

- Sealed Classes
  - Open for classes in the same file, closed (or final) for others.

```
sealed class Card(val suit: String)

class Ace(suit: String) : Card(suit)

class King(suit: String) : Card(suit)

class Queen(suit: String) : Card(suit)

class Jack(suit: String) : Card(suit)
```

149

149

Classes and Objects

- Enum Classes
  - Implement type-safe enum

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

  - Create an enum value

```
var direction:Direction = Direction.WEST
                          Direction.EAST
                          Direction.NORTH
                          Direction.SOUTH
```

150

150

Classes and Objects

- Enum Classes
  - Converting String value to corresponding enum

```
val direction:Direction = Direction.valueOf("EAST")
```

  - Iterating over all the values in an enum

```
for (direction in Direction.values()) {
    println(direction)
}
```

  - Associating a value to enum

```
enum class Direction(val intVal:Int) {
    EAST(1), WEST(2), NORTH(3), SOUTH(4)
}
```

151

151

Classes and Objects

- Data Class
  - Class whose main purpose is to hold data

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

  - Compiler automatically derives the following members from all properties declared in the primary constructor
    - **equals**()/**hashCode**() functions
    - **toString**() function
    - **componentN**() functions corresponding to the properties in their order of declaration
    - **copy**() function.

Classes and Object

- Data Class
  - Requirements
    - 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;

Classes and Objects

- Destructuring Declaration
  - Sometimes it is convenient to *break* an object into a number of variables

```
val (name, age) = person

data class Result(val result: Int, val status: String)
fun function(): Result {
    return Result(101, "Data")
}

val (result, status) = function()
println("$result, $status")
```

154

154

Classes and Object

- Nested Class
  - Classes can be nested in other classes

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo() // == 2
```

  - Nested class does not have access to outer class

155

155

Classes and Objects

- Nested Inner Classes
  - Nested classed can be marked as inner
  - Inner class is able to access members of outer class.
  - Inner classes carry a reference to an object of an outer class:

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo() // == 1
```

156

156

Classes and Objects

- Nested Inner Classes
  - Accessing the superclass of the outer class
    - super keyword qualified with the outer class name: super@Outer

```
class Bar : Foo() {
    override fun f() { /* ... */ }
    override val x: Int get() = 0

    inner class Baz {
        fun g() {
            super@Bar.f()}
    }
}
```

157

157

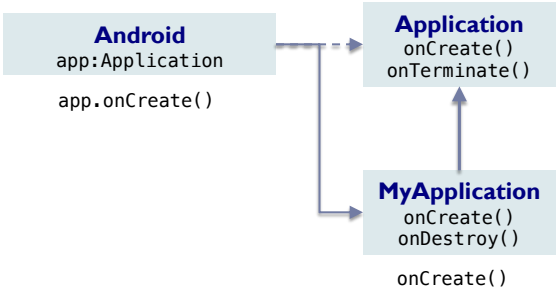
Classes and Objects

- this

```
class A { // implicit label @A
    inner class B { // implicit label @B
        fun foo() { // implicit label @foo
            val b = this@B // B's this
            val a = this@A //this reference of A
        }
    }
}
```


Classes and Objects

- Delegation
  - Customizing using Inheritance

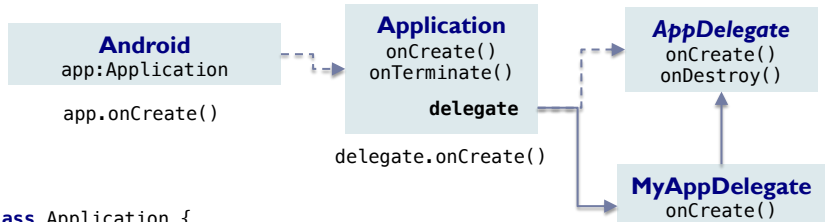



```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()
    }
}
```

```
<manifest>
    <application
        android:name=".MyApplication"
```

Classes and Objects

- Delegation
  - Customization using interface




```
class Application {
    lateinit var delegate:AppDelegate
    fun onCreate() { delegate.onCreate() }
    fun onDestroy() {delegate.onDestroy()}
}

class MyAppDelegate : AppDelegate {
    override fun onCreate() {}
    override fun onDestroy() {}
}
```

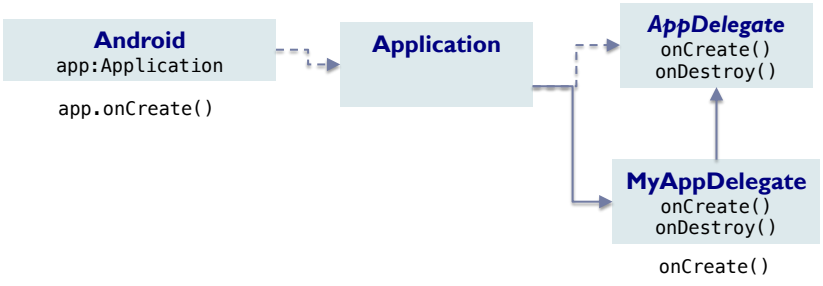
```
var app = Application()
app.delegate = MyAppDelegate()
```

160

160

Classes and Objects

- Delegation
  - Using **by** in kotlin




```
class Application : AppDelegate by MyAppDelegate() {
}
```

161

161

Classes and Objects

- Delegation of Properties
  - Kotlin allows delegation of properties
  - Methods in ReadWriteProperty interface, called when reading or writing to a property

```
operator fun getValue(thisRef: Any?, property: KProperty<*>):T
operator fun setValue(thisRef: Any, property: KProperty<*>, value: T)
```

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.properties/-read-write-property/

Classes and Objects

- Delegation of Properties
  - We can create a class that provides these methods for a type

```
class LoggedString(var content:String) {
    operator fun getValue(thisRef: Any?, property: KProperty<*>):String {
        println("LoggedString = getValue $content")
        return content
    }
    operator fun setValue(thisRef: Any, property: KProperty<*>, value: String) {
        println("LoggedString = $value")
        content = value
    }
}
```

Classes and Objects

- Delegation of Properties
  - Specify that the reading and writing of a property will be handled by a delegate

```
class Person() {
    var name:String by LoggedString("")
}


var person = Person()
person.name = "Raj"



> Task :Delegation_propertyKt.main()
LoggedString = Raj

BUILD SUCCESSFUL in 468ms
Task execution finished 'Delegation_propertyKt.main()'
```


Classes and Objects

- Delegation of Properties
  - Using a factory method to create the delegate

```
fun logged(content:String):LoggedString = LoggedString(content)


class Person() {
    var name:String by logged("")
}
```

Classes and Objects

- Built-in Standard Delegates
  - Lazy

```
val lazyValue: String by lazy {
    println("computed!")
    "Hello"
}
```

  - Observable

```
class Person {
    var name:String by Delegates.observable("") { prop, old, new ->
        println("New name = $new")
    }
}
```

166

166

Classes and Objects

- Built-in Standard Delegates
  - Vetoeable

```
class Counter {
    var count:Int by Delegates.vetoable(0) {_, old, new ->
        new > old
    }
}
```

167

167

Classes and Objects

- Extensions
  - Extend a class with new functionality without having to inherit from the class.
  - Kotlin supports *extension functions* and *extension properties*.

Classes and Objects

- Extension Functions

receiver **Extension function**

```
fun String.plural():String {
    var value = this
    value += "(s)"
    return value
}
```

- Calling Extension function

```
val obj = "Object".plural()
print(obj)
```

Classes and Objects

- Extension Functions
  - same name as member function
    - Member takes precedence

```
class C {
    fun foo() { println("member") }
}

fun C.foo() { println("extension") }

var c = C()
c.foo()
```

Prints => "member"

Classes and Objects

- Nullable Receiver
  - Extensions can be defined with a nullable receiver type
  - Extension function can be called on null reference, and the extension function can check **this == null**

```
fun String?.plural(): String {
    if (this == null) return "null"
    var value = this
    value += "(s)"
    return value
}

val str:String? = null
str.plural()
```

Classes and Objects

- Runtime Type Check
  - is and !is Operators
    - Check whether an object conforms to a given type at runtime

```
var obj = ""
if (obj is String) {
    print("String with length ${obj.length}")
}
if (obj !is String) {
  println("Not String")
}
```

179

179

Classes and Objects

- Smart (Implicit) Type Casting
  - Kotlin compiler keeps track of the is checks for immutable values and inserts casts automatically
    - If the compiler cannot guarantee that the variable cannot change between the check and the usage

```
fun demo(x: Any) {
    if (x is String) {
        // x is automatically cast to String
        print(x.length)
  }
}
```

180

180

Classes and Objects

- Explicit Type Casting
  - **as** operator (unsafe)
    - This cast operator throws an exception if the cast is not possible

```
val y:Int = 500
val x: String = y as String
print(x)
```

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.Integer
cannot be cast to java.lang.String
```

183

183

Classes and Objects

- Explicit Type Casting
  - **as?** Operator (safe)
    - This cast operator returns null on failure

```
val x: String? = y as? String
```

184

184

Functions and Lambda

amit.gulati@gmail.com

190

Functions

- Function types
  - Functions are first class citizens of the Kotlin type system
  - Function type notation
    - All function types have a parenthesized parameter types list and a return type

```
(A, B) -> C

(x: Int, y: Int) -> Point

((Int, Int) -> Int)?

(Int) -> ((Int) -> Unit)
```


191

Functions

- Function Type
  - Example Function Types

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

((Int, Int) -> Int)?

(Int) -> ((Int) -> Unit)
```

192

192

Functions

- Function Type
  - Declaring variables of Function Type

```
var myMathOp:(Int, Int)->(Int)
```

    - myMathOp is a variable that can hold a reference to any function that takes 2 integers as parameters and returns an integer.

193

193

Functions

- Function Type
  - Initializing variable of Function Type
    - Using top level, local, member or extension function

```
//top level
fun multiply(val1:Int, val2:Int) : Int =  val1 * val2

fun main(args: Array<String>) {
    var myMathOp:(Int, Int)->(Int) = ::multiply
    myMathOp(10, 10)
}
```


Functions

- Function Type
  - Initializing variable of Function Type
    - Using top level, local, member or extension function

```
//member function
class MathOps {
    fun addition(val1:Int, val2:Int) : Int = val1 + val2
}

fun main(args: Array<String>) {
    myMathOp = MathOps()::addition
    myMathOp(10, 10)
}
```

Functions

- Anonymous Functions
  - Function without name
  - Body can be either an expression or a block
  - Anonymous Function expression with return value

```
fun(x: Int, y: Int): Int = x + y
```

  - Anonymous Function block with return value

```
fun(x: Int, y: Int): Int {
    return x + y
}
```


Functions

- Anonymous Functions
  - Passing anonymous function to functions as parameter
    - Always passed in parenthesis (no trailing syntax available)

Anonymous function Block

```
var array = arrayOf("Java", "Kotlin", "Swift", "R")
var f = array.filter(fun(val1:String):Boolean {
                        return val1.length >= 4 })
```

Anonymous function Expression

```
var f = array.filter(
       fun(val1:String):Boolean = val1.length >= 4 )
```

Functions

- Anonymous Functions
  - Return type
    - Return type is inferred automatically for anonymous functions with an expression body
    - Specified explicitly (or is assumed to be Unit) for anonymous functions with a block body.

Lambda

- Lambda Expressions
  - Used for creating function literals
  - Helpful when passing executable code to other functions
  - Syntax

```
{ parameter list -> body }
```

```
{ x: Int, y: Int -> x + y }
```

    - Parameters: Specifying the type is optional
    - Body
      - May be a block of code
      - Last expression in the body of is treated as the return value

Lambda

- Lambda Expression
  - Creating a lambda and assigning it to a variable

```
var sum = { a: Int, b: Int ->
              val result = a + b
              println("The result is: $result")
              result }
```

Last expression is returned

  - Executing a lambda

```
val result = sum(20, 20)
```

200

200

Lambda

- Lambda Expression
  - Passing lambda as parameter to a function

```
fun executeMathOp(  val1:Int,
                    val2:Int,
                    op:(Int, Int)->Int) : Int


executeMathOp(10, 10, {x, y -> x * y})
executeMathOp(10, 10, {x, y -> x + y})
```

201

201

Lambda

- Lambda Expression
  - Receiving a lambda as parameter and calling it

```
executeMathOp(10, 10, {x, y -> x * y})


fun executeMathOp(  val1:Int, val2:Int,
                    op:(Int, Int)->Int) : Int {
    return op(val1, val2)
}
```

op = {x, y -> x * y}

202

202

Lambda

- Lambda Expression
  - Passing lambda as parameter to a function
    - Trailing syntax : lambda is the last parameter

```
executeMathOp(10, 10, {x, y -> x * y})

executeMathOp(10, 10) { x, y -> x * y }
```

203

203

Lambda

- Lambda Expression
  - Single parameter named **it**
    - If lambda expression has only one parameter, it can be referred to as **it**

```
fun Array<T>.forEach(action: (T) -> Unit)

var languages = arrayOf("Java", "Kotlin", "Swift", "R")
languages.forEach({ str -> println(str) })

languages.forEach { str -> println(str) }

languages.forEach { println(it) }
```

204

204

Lambda

- Lambda Expression
  - Explicitly returning a value from lambda expression
    - Last line of lambda becomes the implicit return value

```
var sum = { a: Int, b: Int ->
            val result = a + b
            println("The result is: $result")
            result }
val result = sum(20, 20)
```

Last value is returned from lambda expression.
No need for a return keyword.

205

205

Lambda

- Lambda Expression
  - Explicitly returning a value from lambda expression

```
fun executeAction(action:(String) -> Int) {
    println("Calling action with Hello World")
    val result = action("Hello World")
    println("result =  $result")
}

executeAction {
    it.length
}
```

```
> Task :LambdaKt.main()
Calling action with Hello World
result =  11

BUILD SUCCESSFUL in 637ms
```

206

206

Scope Functions

- Scope Functions
  - Execute a block of code (lambda) within the context of an object
  - functions
    - run
    - with
    - apply
    - let
    - also

210

210

Scope Functions

- Scope Functions
  - Difference between the scope functions
    - Refer to the context object
      - Lambda receiver (this)
      - Lambda argument (ii)

```
val str = "Hello"
str.run {
    println("The receiver string length: $length ")
    //println("The receiver string length: ${this.length}")
}

str.let {
    println("The receiver string's length is ${it.length}")
}
```

211

Scope Functions

- Scope Functions
  - Difference between the scope functions
    - Return value

```
val str:String? = "Hello"
val length = str?.let { it.length }

var str1 = "Hello"
val str2 = str1.apply {
    str1 + "World !!"
}
```

212

Scope Functions

- Scope Functions
  - Context, Return

Function	Context	Return	Extension
run	this	lambda result	yes
with	this	lambda result	no
apply	this	context	yes
let	argument name or **it**	lambda result	yes
also	argument name or **it**	context	

Scope Functions

- **apply**
  - Extension function
  - Context object is the receiver and can be accessed using **this**
  - Returns the receiver.

```
val file = File("file.txt")
file.setReadable(true)
file.setWritable(true)
file.setExecutable(false)
```

```
val file = File("file.txt").apply {
    setReadable(true)
    setWritable(true)
    setExecutable(false)
}
```

Scope Functions

- **let**
  - Extension functiob
  - Context passed as single argument to lambda and can be accessed using **it**
  - Returns the result of the lambda

```
val sometimesNull =
       if (Random().nextBoolean()) "not null" else null

sometimesNull?.let {
    println("It was $it this time")
}
```


Scope Functions

- **with**
  - Not an extension funcyion
  - Context object is the receiver and can be accessed using **this**
  - Returns the result of the lambda

```
val myTurtle = Turtle()
with(myTurtle) { //draw a 100
pix square
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0)
    }
    penUp()
}
```

Scope Functions

- **run**
  - Context object is the receiver and can be accessed using **this**
  - Returns the result of the lambda

```
val file = File("file.txt")
val containsKotlin = menuFile.run {
    this.readText().contains("Kotlin")
}
```


Scope Functions

- **also**
  - Extension function
  - Context passed as single argument to lambda and can be accessed using **it**
  - Returns the receiver

```
val numbers = mutableListOf("one", "two", "three")
numbers.also { println("numbers before adding new one: $it") }
        .add("four")
```

Scope functions

- Choose the correct scoping function
  - Executing a lambda on non-null objects: **let**
  - Object configuration: **apply**
  - Object configuration and computing the result: **run**
  - Additional effects: **also**
  - Grouping function calls on an object: **with**

Standard Library

- Collection Filtering Functions
  - **drop**
    - removes first n elements from the collection.

```
val numbers = listOf(1, 2, 3, 4, 5)
val dropped = numbers.drop(2)
```

  - **filter**
    - apply a predicate function to the collection

```
val numbers = listOf(1, 2, 3, 4, 5)
val smallerThan3 = numbers.filter { n -> n < 3 }
```

Standard Library

- Collection Filtering Functions
  - **take**
    - Takes the first n elements from collection.

```
val numbers = listOf(1, 2, 3, 4, 5)
val first2 = numbers.take(2)
```

221

221

Standard Library

- General Function
  - **any**
    - returns true if the supplied predicate function matches any of the collection items

```
val numbers = listOf(1, 2, 3, 4, 5)
val hasEvens = numbers.any { n -> n % 2 == 0 }
```

  - **count**
    - returns the count of items that match the given predicate function

```
val numbers = listOf(1, 2, 3, 4, 5)
val evenCount = numbers.count { n -> n % 2 == 0 }
```

222

222

Standard Library

- General Function
  - **forEach**
    - iterates over the collection and calls the given action function on each item

```
val numbers = listOf(1, 2, 3, 4, 5)
numbers.forEach { n -> println(n) }
```


Standard Library

- Transformation Function
  - **map**
    - Applies the given transform function on each item in the collection

```
val numbers = listOf(1, 2, 3, 4, 5)
val strings = numbers.map { n -> n.toString() }
```

Idiomatic Kotlin

amit.gulati@gmail.com

225

Idiomatic Kotlin

- Use Builder to create collections
  - Read only List

```
val list = listOf("a", "b", "c")
```

  - Read only Map

```
val map = mapOf("a" to 1, "b" to 2, "c" to 3)
```

  - Mutable List and Map

```
val mutableList = mutableListOf("a", "b", "c")

val mutableMap = mutableMapOf(   "a" to 1,
                                 "b" to 2,
                                 "c" to 3)
```

226

226

Idiomatic Kotlin

- Use Range Operators instead of comparison pairs

```
fun isLatinUppercase(c: Char) =
    c >= 'A' && c <= 'Z'
```

```
fun isLatinUppercase(c: Char) =
    c in 'A'..'Z'
```

227

Idiomatic Kotlin

- Use when as expressions

```
fun parseEnglishNumber(number: String): Int? {
    when (number) {
        "one" -> return 1
        "two" -> return 2
        else -> return null
    }
}
```

```
fun parseEnglishNumber(number: String) = when (number) {
    "one" -> 1
    "two" -> 2
    else -> null
}
```

228

228

Idiomatic Kotlin

- Use if as expression

```
fun checkPositive(value:Int):Boolean {
    if (number > 0) {
       return true
    } else {
       return false
    }
}
```

```
fun checkPositive(value:Int):Boolean {
    return if (number > 0) {
        true
    } else {
       false
    }
}
```

229

229

Idiomatic Kotlin

- Use try as expression

```
fun tryParseInt(number: String): Int? {
    try {
        return Integer.parseInt(number)
    } catch (e: NumberFormatException) {
        return null
    }
}
```

```
fun tryParseInt(number: String) =
    try {
        Integer.parseInt(number)
    } catch (e: NumberFormatException) {
        null
    }
```

230

230

Idiomatic Kotlin

- Safe access to nullable types

```
val files = File("Test").listFiles()
if( files != null ) {
    println(files.size)
}
```

  - If not null

```
val files = File("Test").listFiles()
println(files?.size)
```

  - If not null else

```
val files = File("Test").listFiles()
println(files?.size ?: "empty")
```


Idiomatic Kotlin

- Safe access to nullable types
  - Execute if not null

```
fun printName(name:String?){
    name?.let {
        println(it)
    }
}
```

Idiomatic Kotlin

- Calling multiple methods on an object instance using **with**

```
class Turtle {
    fun penDown() {}
    fun penUp() {}
    fun turn(degrees: Double) {}
    fun forward(pixels: Double) {}
}

val myTurtle = Turtle()
with(myTurtle) {
    penDown()
    for(i in 1..4) {
        forward(100.0)
        turn(90.0) }
    penUp()
}
```

233

233

Idiomatic Kotlin

- Data Class
  - Use Data class
    - POJO
    - Returning multiple types from function
    - Etc.

```
data class Customer(val name: String, val email: String)
```

    - Functionality added
      - Equals()
      - hashCode()
      - toString()
      - copy()
      - component1(), component2()…….

234

234

Idiomatic Kotlin

- Default values for function parameter
  - Avoid overloading functions to provide different number of parameters

```
fun foo(a: Int, b: String) { ... }
fun foo(a: Int) { ... }
```

```
fun foo(a: Int, b: String = "") { ... }
```


Idiomatic Kotlin

- String Interpolation vs Concatenation

```
println("Name " + $name)
```

```
println("Name $name")
```

Idiomatic Kotlin

- Instance Check

```
fun takeAction(animal:Any) {
    if( (animal as? Dog) != null) {
        print("Animal is Dog")
    } else if( (animal as? Cat) != null) {
        print("Animal is Cat")
    }
}
```

```
fun takeAction(animal:Any ){
    when (animal) {
        is Dog -> print("Animal is Dog")
        is Cat -> print("Animal is cat")
    }
}
```

237

237

Idiomatic Kotlin

- Function Expressions

```
fun celciusToFahrenheit(celsius:Float):Float {
    return (celsius * 1.8f) + 32.0f
}
```

```
fun celciusToFahrenheit(celsius:Float) =
            (celsius * 1.8f) + 32.0f
```

238

Idiomatic Kotlin

- Don't create classes just to put function

```
class StringUtils {
    companion object {
        fun isPhoneNumber(s: String) =
            s.length == 7 && s.all { it.isDigit() }
    }
}
```

```
object StringUtils {
    fun isPhoneNumber(s: String) =
        s.length == 7 && s.all { it.isDigit() }
}
```

```
fun isPhoneNumber(s: String) =
    s.length == 7 && s.all { it.isDigit() }
```

239

239

Idiomatic Kotlin

- Use extension function where possible

```
fun isPhoneNumber(s: String) =
    s.length == 7 && s.all { it.isDigit() }
```

```
fun String.isPhoneNumber() =
    length == 7 && all { it.isDigit() }
```

240

240

Idiomatic Kotlin

- Consider extracting non-essential API of classes into extensions

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

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

```
class Person( val firstName: String,
              val lastName: String)

//property extension
val Person.fullName: String
    get() = "$firstName $lastName"
```

241

241

Idiomatic Kotlin

- Use **lateinit** for properties that cannot be initialized in a constructor

```
class MyTest {
    class State(val data: String)

    private var state: State? = null

}
```

```
class MyTest {
    class State(val data: String)

    private lateinit var state: State

}
```

242

242

Idiomatic Kotlin

- Use data classes to return multiple values

```
data class NamedNumber(
    val number: Int,
    val name: String
)

fun namedNum() =
    NamedNumber(1, "one")

fun main(args: Array<String>) {
    val (number, name) = namedNum()
}
```

243

243

Idiomatic Kotlin

- Use **apply** for object initialization

```
fun createLabel(): JLabel {
    val label = JLabel("Foo")
    label.foreground = Color.RED
    label.background = Color.BLUE
    return label
}
```

```
fun createLabel() = JLabel("Foo").apply {
    foreground = Color.RED
    background = Color.BLUE
}
```

244

244

Idiomatic Kotlin

- Easy lazy properties

```
private var _os: String? = null
val os: String
    get() {
        if (_os == null) {
            println("Computing...")
            _os = System.getProperty("os.name") +
                    " v" + System.getProperty("os.version") +
                    " (" + System.getProperty("os.arch") + ")"
        }
        return _os!!
    }
```

```
val os: String by lazy {
            println("Computing...")
            System.getProperty("os.name") +
                    " v" + System.getProperty("os.version") +
                    " (" + System.getProperty("os.arch") + ")"
    }
```

245

245

Coroutines

249

249

Coroutines

- Introduction
  - Great way to create concurrent non blocking code
  - Suspendable computation instead of blocking thread
  - Compiler manages the execution state of coroutines
  - Highly Scalable

Coroutine

- Support
  - Part of the language since 1.3
  - Helper functions part of the standard library
  - Google also provides a library for using coroutines win Android
  - Add dependencies in build.gradle file

```
org.jetbrains.kotlinx:kotlinx-coroutines-core:1.4.2'
org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9
```

Coroutine

- Core Concepts
  - Suspend functions
  - Coroutine Builders
  - Dispatchers
  - Context
  - Job

Coroutine

- Suspend Functions
  - Marked with the keyword **suspend**
  - Execution of suspendable function can be put aside without blocking the thread
    - Thread can execute other code
  - Can only be called from
    - Another suspend function
    - Context of Co-routine Builder

Coroutine

- Suspend functions
  - Sequentially executing tasks in a function (subroutine)

```
fun task1() {
    println("start task1 in Thread ${Thread.currentThread()}")
    println("end task1 in Thread ${Thread.currentThread()}")
}
fun task2() {
    println("start task2 in Thread ${Thread.currentThread()}")
    println("end task2 in Thread ${Thread.currentThread()}")
}

fun main(args: Array<String>) {
    println("start")
    task1()
    task2()
    println("done")
}
```

254

254

Coroutine

- Suspend functions
  - Sequentially executing tasks in a coroutine

```
fun main(args: Array<String>) {
    println("start")
    runBlocking {
        task1()
        task2()
    }
    println("done")
}
```

  - runBlocking function creates a coroutine.

255

255

Coroutines

- Suspend functions
  - Concurrent executing tasks in a coroutine

```
suspend fun task1() {
    println("start task1 in Thread ${Thread.currentThread()}")
    yield()
    println("end task1 in Thread ${Thread.currentThread()}")
}
suspend fun task2() {
    println("start task2 in Thread ${Thread.currentThread()}")
    yield()
    println("end task2 in Thread ${Thread.currentThread()}")
}

fun main(args: Array<String>) {
    runBlocking {
        launch {task1()}
        launch {task2()}
    }
}
```

256

256

Coroutines

- Coroutine Context
  - Coroutines run in the same thread in which they are created.
  - Vary the context and the thread of execution
  - Passed to the coroutine builder
  - Decide which thread to start or resume a coroutine on
    - Dispatchers.Default
    - Dispatchers.IO
    - Dispatchers.Main
    - Dispatchers.Unconfined

257

257

Coroutines

- Coroutine Builders
  - **runBlocking**
    - Creates a new coroutine for executing a lambda as suspend function

```
runBlocking {
    task1()
    task2()
}
```

task1, task2 may be suspend functions

    - Blocks the current thread until its completion
    - Coroutine runs in the same context as the thread in which runBlocking is called.
    - A different context can be provided as parameter when calling

```
runBlocking(Dispatchers.Default){
```

258

Coroutines

- Coroutine Builders
  - **runBlocking**
    - Creates a new coroutine for executing a lambda as suspend function

```
runBlocking {
    task1()
    task2()
}
```

task1, task2 may be suspend functions

    - Blocks the current thread until its completion
    - Coroutine runs in the same context as the thread in which runBlocking is called.
    - A different context can be provided as parameter when calling

```
runBlocking(Dispatchers.Default){
```

259

Coroutines

- Coroutine Builders
  - **launch**
    - Creates a new coroutine for executing a lambda as suspend function

```
runBlocking(Dispatchers.Default){
    val job1 = launch{task1()}
    val job2 = launch {task2()}
}
```

    - Requires a CoroutineScope to be called
    - Returns a **Job** object
      - **Job** object can be used to await termination of coroutine
    - No way to return a result from the coroutine
    - Coroutine runs in the same context as the thread in which launch is called.
    - A different context can be provided as parameter when calling

```
val job1 = launch(Dispatchers.Default){ task1() }
```


Coroutines

- Coroutine Builders
  - **launch**
    - Job Interface

```
public interface Job {
    public val isActive: Boolean
    public val isCompleted: Boolean
    public val isCancelled: Boolean
    public fun cancel()
}
```

Coroutines

- Coroutine Builders
  - **async**
    - Creates a new coroutine for executing a lambda as suspend function

```
val result = async {
    Runtime.getRuntime().availableProcessors()
}
```

    - Requires a CoroutineScope to be called
    - Returns a Deferred<T> object
    - Coroutine runs in the same context as the thread in which launch is called.
    - A different context can be provided as parameter when calling

```
async(Dispatchers.Default) {
```


Coroutines

- Coroutine Builders
  - **async**

```
runBlocking {
    val firstSum = async {
        println(Thread.currentThread().name)
        add(2, 2)
    }
    val secondSum = async {
        println(Thread.currentThread().name)
        add(3, 4)
    }
    println("Awaiting concurrent sums...")
    val total = firstSum.await() + secondSum.await()
    println("Total is $total")
}

suspend fun add(x: Int, y: Int): Int {
    delay(Random.nextLong(1000L))
    return x + y
}
```

1

Coroutines

- Coroutine Builders
  - **withContext** function to replace async/await

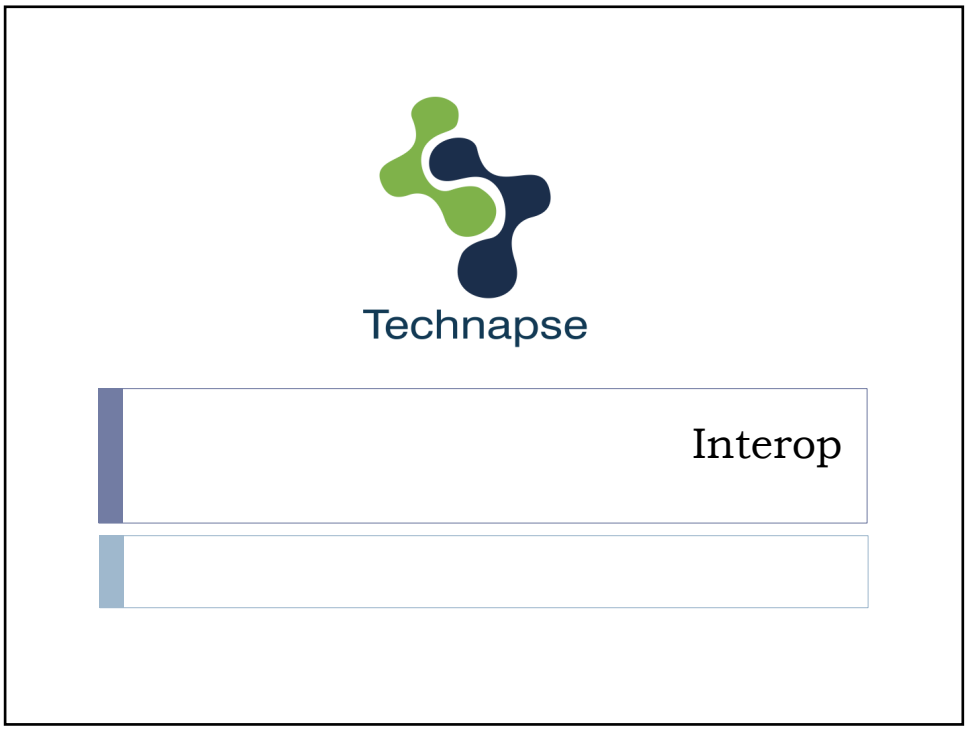
```
async(Dispatchers.IO) {
    println("Retrieving data on ${Thread.currentThread().name}")
    delay(100L)
    "asyncResults"
}.await()


withContext(Dispatchers.IO) {
    println("Retrieving data on ${Thread.currentThread().name}")
    delay(100L)
    "withContextResults"
}
```


Coroutines

- CoroutineScope Builder
  - **coroutineScope**
    - suspending function that waits until all included coroutines finish before exiting.

```
coroutineScope {
    for (i in 0 until 10) {
        launch {
            delay(1000L - i * 10)
            print("$i ")
        }
    }
}
```

    - Requires a CoroutineScope to be called



266

InterOp

- Java from Kotlin
  - Getters and Setters
    - Represented as properties in Kotlin

Java

```java
public class RectangleJava {
    private int width;
    public int getWidth() {
       return width;
    }
    public void setWidth(int width) {
       this.width = width;
    }
}
```

Kotlin

```kotlin
var rect = RectangleJava()
println(rect.width)
rect.width = 100
```

267

InterOp

- Java from Kotlin
  - Getters and Setters
    - If the Java class only has a setter, it will not be visible as a property in Kotlin,
      - Set-only properties are not supported in Kotlin at this time.

268

InterOp

- Java from Kotlin
  - Methods returning void
    - Java method returning void, will return Unit in Kotlin

Java

```
public void printRectangle() {
    System.out.print("Rectangle(" + width +
                     ", " + height + ")");
}
```

Kotlin

```
var rect = RectangleJava()
var result:Unit = rect.printRectangle()
```

269

269

InterOp

- Java from Kotlin
  - Escaping for Java identifiers that are keywords in Kotlin
    - Kotlin keywords that are valid identifiers in Java:
      - **in**, **object**, **is**, etc
    - Escape the keyword using (‘) character.

```
foo.`is`(bar)
```


InterOp

- Null Safety and Platform Types
  - Java declarations are treated specially in Kotlin and called *platform types*.
  - Platform Types
    - Null-checks are relaxed for such types
      - Kotlin does not issue nullability errors at compile time for method calls.

InterOp

- Null Safety and Platform Types
  - Compiler infers the appropriate type

```
public class RectangleJava {
    public static RectangleJava createRectangle() {
        return new RectangleJava();
    }
}


var rect1 = RectangleJava.createRectangle()
rect1.printRectangle()
```

  - Type inferred for rect1 is **RectangleJava!**

272