fun() Talk

Exploring Kotlin Functions

@n8ebel

Kotlin ❤ Functions

Easy to Get Started

- Can use the IDE conversion tool
- Can try online (https://try.kotlinlang.org)
- Easy to transfer existing knowledge

Flexible & Convenient

- Parameter & type flexibility
- Variations in scoping
- Variety of modifiers

Freedom to Reimagine

The flexibility & functionality of functions allow us to break away from traditional java conventions and reimagine how we build our projects

Let's Have Some fun()

From Methods to Functions

Hello Java Method

```
void helloFunctions() {
  System.out.println("Yay, Functions");
}
```

- return type
- name
- method body

java method converted to kotlin

```
fun helloFunctions() {
  println("Yay, Functions")
}
```

- adds the fun keyword
- no explicit return type
- same name

Further Simplification

```
fun helloFunctions() = println("Yay, Functions")
```

So What's Left?

Seems pretty straightforward, what else is there to know?

A Lot

Let's Build On What We Know

Parameter & Type Freedom

- Default parameters
- Named parameters
- Return types
  - when can we omit?
  - when can we infer?
- Generic functions

Parameters

```
fun helloFunctions(excitingThing:String) {
  println("Yay, " + excitingThing)
}

helloFunctions("functions")
// outputs "Yay, functions"
```

Parameters

```
fun helloFunctions(exclamation:String, excitingThing:String) {
  println(exclamation + ", " + excitingThing)
}

helloFunctions("Yay", "functions")
// outputs "Yay, functions"
```

Now It Gets Interesting

Default Parameter Values

```
fun helloFunctions(exclamation:String, excitingThing:String = "functions") {
  println(exclamation + ", " + excitingThing)
}

helloFunctions("Yay", "functions")
// outputs "Yay, functions"

helloFunctions("Yay")
// outputs "Yay, functions"
```

Function parameters can have default values, which are used when a corresponding argument is omitted

Default Parameter Values

- allows us the flexibility of overloads without the verbosity of writing them
- help document the function contract by indicating what "sensible defaults" might be

Default Parameters & Java

Java doesn't have default parameter values

- must specific all parameter values when calling from Java
- can use `@JvmOverloads` to generate overloads for each parameter
- generated overloads will used the specified default values

Named Arguments

Improve readability of function invocations

```
helloFunctions("functions", "functions")
```

- How do we know which value is correct?

Named Arguments

Much easier to understand with named arguments

```
helloFunctions(exclamation = "yay!", excitingThing = "functions")
```

Named Arguments

Modify order of passed parameters by using named arguments

```
fun helloFunctions(exclamation:String, excitingThing:String = "functions") {
  println(exclamation + ", " + excitingThing)
}

helloFunctions("Hooray", "functions")
helloFunctions("Hooray")
helloFunctions(excitingThing = "functions", exclamation = "Hooray")
// all output "Hooray, functions"
```

Named Arguments

There are limitations to how named & positioned arguments are used

- once an argument name is specificed, all subsequent arguments must be named as well

Named Arguments

```
helloFunctions("hooray", "Droidcon Boston")
helloFunctions("hooray", excitingThing = "Droidcon Boston")
// both output "hooray, Droidcon Boston"

helloFunctions(excitingThing = "Droidcon Boston", "hooray")
// error: Mixing named and positioned arguments not allowed
```

Variable Number of Arguments

We can define a parameter to accept a variable number of arguments `T`

- use the `vararg` keyword
- the `vararg` param is then treated as an array of type T
- default value must now be an array

Variable Number of Arguments

```
fun helloFunctions(exclamation:String, vararg excitingThings:String) {
  for(excitingThing in excitingThings) {
    println(exclamation + ", " + excitingThing)
  }
}

helloFunctions("yay!", "Droidcon Boston", "Kotlin", "Android")
// outputs:
// yay!, Droidcon Boston
// yay!, Kotlin
// yay!, Android
```

Variable Number of Arguments

Typically, a `vararg` parameter will be the last one

Can be used in any order if:
- other parameters are called using named argument syntax
- last parameter is a function passed outside the parentheses

Variable Number of Arguments

This works great

```
helloFunctions("yay!", "Droidcon Boston", "Kotlin", "Android")
helloFunctions("Droidcon Boston", "Kotlin", "Android", exlamation = "yay!")

// both output:
// yay!, Droidcon Boston
// yay!, Kotlin
// yay!, Android
```

Variable Number of Arguments

This works

```
helloFunctions("Droidcon Boston",  "Kotlin", "Android")
// output:
// "Droidcon Boston, Kotlin"
// "Droidcon Boston, Android"
```

Variable Number of Arguments

This won't compile

```
helloFunctions("Droidcon Boston", exlamation = "yay!", "Kotlin", "Android")
// error: "no matching function"
```

Variable Number of Arguments

Use "spread" operator to pass an existing array of values

```
val thingsToBeExcitedAbout = arrayOf("Droidcon Boston", "Kotlin", "Android")
helloFunctions("yay!", *thingsToBeExcitedAbout)

// output:
// yay!, Droidcon Boston
// yay!, Kotlin
// yay!, Android
```

Variable Number of Arguments

"Spreading" can be used alone, or with other passed varargs as well

```
helloFunctions("yay!", "coffee", *thingsToBeExcitedAbout)
helloFunctions("yay!", *thingsToBeExcitedAbout, "coffee")
```

- input array to the vararg parameter is handled in order

Return Types

What is the return type?

```
fun helloFunctions(exclamation:String, excitingThing:String="functions") {
  println(exclamation + ", " + excitingThing)
}
```

If a function does not return any useful value, its return type is Unit

Return Types

These are equivalent

```
fun helloFunctions(exclamation:String, excitingThing:String="functions") : Unit {
  println(exclamation + ", " + excitingThing)
}

fun helloFunctions(exclamation:String, excitingThing:String="functions") {
  println(exclamation + ", " + excitingThing)
}
```

Return A Non-Unit Type

Functions with block body require explicit return type & call for non-Unit functions

```
fun helloFunctions(exclamation:String, excitingThing:String="functions") : String {
  return exclamation + ", " + excitingThing
}
```

Return A Non-Unit Type

Can infer return type for single-expression functions

```
fun helloFunctions(exclamation:String, excitingThing:String="functions")
    = exclamation + ", " + excitingThing
```

Generic Functions

Like classes, functions may have generic type parameters

```
public inline fun <T> Iterable<T>.filter(predicate: (T) -> Boolean): List<T> {
    return filterTo(ArrayList<T>(), predicate)
}

public fun <T> Iterable<T>.toHashSet(): HashSet<T> {
    return toCollection(HashSet<T>(mapCapacity(collectionSizeOrDefault(12))))
}

listOf(2,4,6,8).filter{ ... }
setOf(2,4,6).toHashSet()
```

concise
convenient
flexible

What Next?

Let's explore some variations on how you can create and use functions

Variations In Scope

- Top-level
- Member functions
- Local
- CompanionObject
- Extension functions

Top-Level functions

- Not tied to a class
- Defined within a Kotlin file
- Belong to their declared file's package
- Import to use within other packages

Top-Level Function Patterns

- Replace stateless classes filled with static methods
- Swap your "Util" or "Helper" classes with functions

Top-Level Function Considerations

- Not truly removing classes
- Generated as a public static method on a class using a special convention
- <function's file name>Kt.java

Top-Level Function Considerations

Inside Logging.kt

```
package logging

fun log(error:Throwable) {...}
```

Top-Level Function Considerations

Call from Kotlin

```
log(Throwable("oops"))
```

Top-Level Function Considerations

Generated Code

```
public class LoggingKt {
  public static void log(Throwable error) {...}
}
```

Top-Level Function Considerations

Call from Java

```
LoggingKt.log(new Throwable("oops"))
```

Top-Level Function Considerations

Can override the generated class/file name

- Add `@file:JvmName(<desired class name>)` to function's file
- Must be before the declared package

Top-Level Function Considerations

Inside Logging.kt

```
@file:JvmName("LoggingFunctions")
package logging

fun log(error:Throwable) {...}
```

Top-Level Function Considerations

Call from Java

```
LoggingFunctions.log(new Throwable("oops"))
```

Top-Level Function Summary

- Function declared in a file outside of any class
- Can replace stateless helper/util classes
- Can override generated class name to improve Java interop

Member Functions

- Function on a class or object
- Like a Java method
- Have access to private members of the class or object

Member Functions

```
class Speaker() {
    fun giveTalk() { ... }
}

// create instance of class Speaker and call giveTalk()
Speaker().giveTalk()
```

Member Function Considerations

- Default arguments can't be changed in overridden methods
- If overriding a method, you must omit the default values

Local Functions

Functions inside of functions

- Create a function that is scoped to another function
- Useful if your function is only ever called from another function

Local Functions

- Declare like any other function, just within a function
- Have access to all params and variables of the enclosing function

Local Functions

Why would you want this?

- Clean code
- Avoids code duplication
- Avoid deep chains of function calls

Local Functions

```
fun parseAccount(response:AccountResponse) : Account {
  ...

  val hobbies = response.getField("hobbies").map{
    val key = it.getField("key")
    Hobby(key)
  }

  val favoriteFoods = response.getField("favorite_foods").map{
    val key = it.getField("key")
    Food(key)
  }
}
```

Local Functions

```
fun parseAccount(response:AccountResponse) : Account {
  fun parseKey(entity:ResponseEntity) = entity.getField("key")

  ...

  val hobbies = response.getField("hobbies").map{
    val key = parseKey(it)
    Hobby(key)
  }

  val favoriteFoods = response.getField("favorite_foods").map{
    val key = parseKey(it)
    Food(key)
  }
}
```

Local Function Considerations

- Local function or private method?
- Is the logic going to be needed outside the current function?
- Does the logic need to be tested in isolation?
- Is the enclosing function still readable?

Companion Objects

- No static method/functions in Kotlin
- Recommended to use top-level functions instead
- What if you need access to private members of an object?

Companion Objects

- Want to create a factory method?
- Define a member function on a companion object to gain access to private members/constructors

Companion Objects

```
class Course private constructor(val key:String)

// won't work
// can't access the private constructor
fun createCourse(key:String) : Course {
    return Course(key)
}
```

Companion Objects

```
class Course private constructor(val key:String) {
    companion object {
        fun createCourse(key:String) : Course {
            return Course(key)
        }
    }
}

// can then call the factory method
Course.createCourse("somekey")
```

Companion Object Function Considerations

Java usage is ugly

```
// from Java
Course.Companion.createCourse("somekey")
```

Companion Object Function Considerations

```
class Course private constructor(val key:String) {
    companion object Factory {
        fun createCourse(key:String) : Course {
            return Course(key)
        }
    }
}

// from Java
Course.Factory.createCourse("somekey")
```

different scopes for different use cases

Variations

Variations

- `infix`
- `extension`
- higher-order
- `inline`

infix

- `infix` keyword enables usage of infix notation
- What is infix notation?
- Can omit the dot & parentheses for the function call
- `"key" to "value"`

infix

- Must be a member function or extension function
- Must take a single, non-varargs, parameter with no default value

infix

```
class ConferenceAttendee {
    infix fun addInterest(name:String){...}
}

// call the function without dot or parentheses
val attendee = ConferenceAttendee()
attendee addInterest "Kotlin"
```

infix

- Provides a very clean, human-readable syntax
- Core building block of custom DSLs

infix

```
"hello" should haveSubstring("ell")
"hello" shouldNot haveSubstring("olleh")
```

https://github.com/kotlintest/kotlintest

Extension Functions

- Extend the functionality of an existing class
- Defined outside the class
- Used as if they were a member of a class

Why Extension Functions?

- Clean-up or extend classes & apis you don't control
- Remove helper classes & simplify top-level functions

Extension Functions

```
// add a new function to the View class
fun View.isVisible() = visibility == View.VISIBLE

yourView.isVisible()
```

Extension Functions

```
fun showToast(
    context: Context,
    msg:String,
    duration: Int = Toast.LENGTH_SHORT) {

    Toast.makeText(context, msg, duration).show()
}

showToast(context, "Toast!")
```

Extension Functions

```
fun Context.showToast(
  msg: CharSequence,
  duration: Int = Toast.LENGTH_SHORT) {

  Toast.makeText(this, msg, duration).show()
}

context.showToast("Toast!")
```

Extension Function Considerations

- How are these generated under the hood?
- How are these called from Java?

Extension Function Considerations

- Generated as static methods that accept the receiver object as it's first argument
- Default behavior is to use <filename>Kt.<functionName>

Extension Function Considerations

```
// ContextExtensions.kt

fun Context.showToast(...) { ... }

// when called from Java
ContextExtensionsKt.showToast(context, "Toast!");
```

Higher-Order Functions

- Functions that take, or return, other functions
- Can be lambda or function reference
- Many examples in the Kotlin standard library `apply`, `also`, `run`

Higher-Order Functions

- Enable interesting patterns & conventions
- Support functional programming
- Can cleanup setup/teardown patterns such as shared prefs

Higher-Order Functions

```
fun getScoreCalculator(level:Level) {
  return when (level) {
    Level.EASY -> { state:ScoreState -> state.score * 10 }
    Level.HARD -> { state:ScoreState -> state.score * 5 * state.accuracy }
  }
}
```

Higher-Order Functions

```
val predicate = { number:Int -> number > 5 }
listOf(2,4,6,8).filter(predicate)
```

Higher-Order Functions

```
fun filterTheList(value:Int) = value > 5
listOf(2,4,6,8).filter(::filterTheList)
```

Higher-Order Functions

If the last parameter of a function is a function, you can omit the parentheses

```
listOf(2,4,6,8).filter{ number -> number > 5 }
```

Higher-Order Functions

```
public inline fun <R> synchronized(lock: Any, block: () -> R): R {
    monitorEnter(lock)
    try {
        return block()
    }
    finally {
        monitorExit(lock)
    }
}

// call from Kotlin
synchronized(database) {
  database.prePopulate()
}
```

Higher-Order Function Performance

"Using higher-order functions imposes certain runtime penalties"

- Extra class created when using lambda
- If lambda captures variables, extra object created on each call

inline

- Helps solve higher-order function performance hits
- Body of the inlined function is substituted for invocations of the function

inline

```
inline fun <T> synchronized(lock: Lock, action: () -> T): T {
    lock.lock()
    try {
        return action()
    }
    finally {
        lock.unlock()
    }
}

// call from Kotlin
synchronized(Lock()) {...}
```

inline

```
// sample usage
fun inlineExample(l:Lock) {
  println("before")
  synchronized(l) {
    println("action")
  }
  println("after")
}
```

inline

With `inline` the generated code is equivalent to this

```
// resulting code
fun inlineExample(l:Lock) {
  println("before")
  lock.lock()
  try {
      println("action")
  }
  finally {
      lock.unlock()
  }
  println("after")
}
```

Android Reimagined

These features enabled us to rethink how we build our apps

Fewer Helper Classes

- `ContextHelper`, `ViewUtils`
- Replace with
  - top-level functions
  - extension functions

Less Boilerplate

```
fun doTheThingSafely(theThing:() -> Unit) {
    try {
      theThing()
    } catch(error:Throwable) {
      // handle error
    }
}
```

Upgrade Our Apis

Can use extensions, default params, etc to cleanup common apis

- Now seeing community supported examples of this
- Android KTX: https://github.com/android/android-ktx
- Anko: https://github.com/Kotlin/anko

Android KTX

```
sharedPreferences.edit()
        .putString("key", "without ktx")
        .putBoolean("isLessBoilerplate", false)
        .apply()

sharedPreferences.edit {
    putString("key", "with ktx")
    putBoolean("isLessBoilerplate", true)
}
```

Cleaner Syntax

```
fun log(msg:String) {...}

inline fun runOnBackgroundThread(action:() -> Unit) { ... }
```

- More fluent syntax
- Simplify test mocking
- Avoids extra classes

DSLs

```
val articleBuilder = ArticleBuilder()
  articleBuilder {
    title = "This is the title"
    addParagraph {
      body = "This is the first paragraph body"
    }
    addParagraph {
      body = "This is the first paragraph body"
      imageUrl = "https://path/to/url"
    }
  }
```

- https://proandroiddev.com/kotlin-dsl-everywhere-de2994ef3eb0

DSLs

DSL examples

- https://github.com/gradle/kotlin-dsl
- https://github.com/kotlintest/kotlintest
- https://github.com/Kotlin/anko/wiki/Anko-Layouts
- https://kotlinlang.org/docs/reference/type-safe-builders.html

Kotlin functions provide flexibility & freedom in how you build your apps

Go, and Have fun()

- Easy to get started
- Can build your understanding and usage of functions over time
- Enables you to rethink how you build your applications

Ready to Learn More?

- https://engineering.udacity.com
- https://n8ebel.com/tag/kotlin

- Udacity Course: https://www.udacity.com/course/kotlin-for-android-developers--ud888
- Kotlin In Action

Thanks For Coming

Let's Continue the Discussion

```
with("n8ebel").apply {
   Twitter
   .com
   Medium
   Instagram
   Facebook
   GitHub
}
```