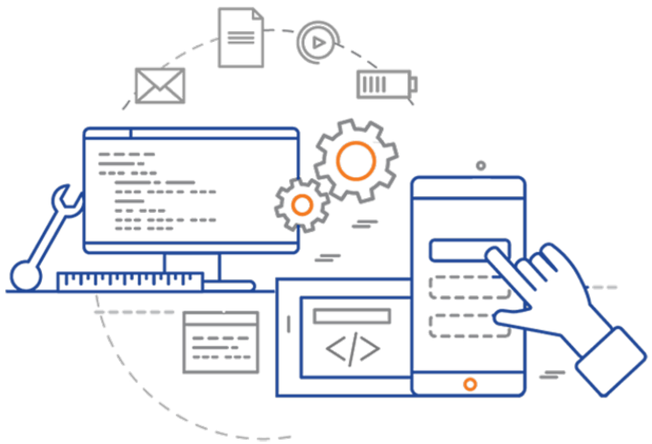


CS 398: Application Development

Functional Kotlin

Function types; Lambdas; Collections; Sequences

Functional Programming

Functional Programming (FP) is a declarative programming style where programs are constructed by **compositing functions together**. As much as possible, computation is expressed as a series of functions that return values.

- Functional programming treats **functions as first-class citizens**.
- Functional programming also **specifically avoids mutation** and **side effects** (inadvertent changes to state).

Don't worry, we're not bringing back Racket.

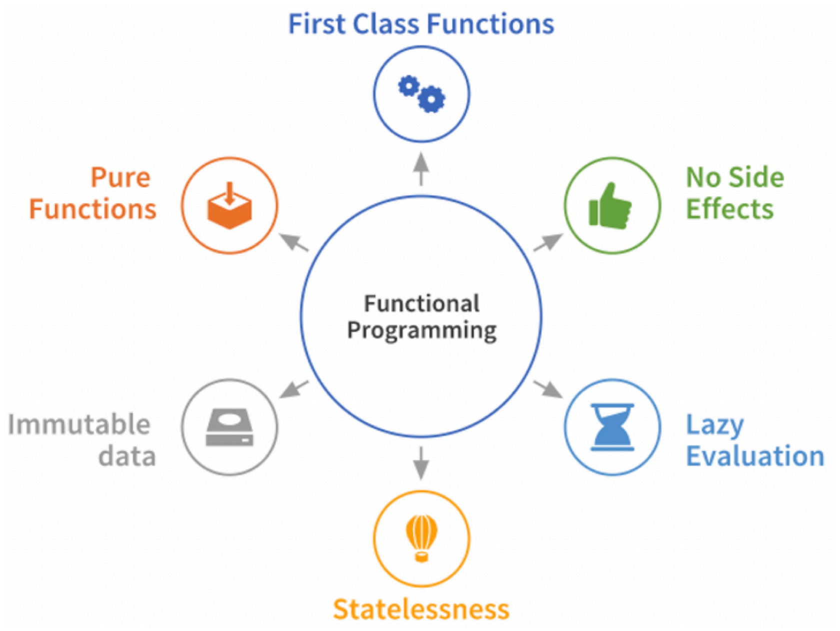
FP Paradigm

First-class functions means that functions are treated as *any other type*. We can pass them as to another function as a parameter, return functions from other functions, and assign functions to variables.

Pure functions are functions that have no side effects. More formally, the return values of a pure function are identical for identical arguments (i.e. they don't depend on any external state).

Immutable data means that we do not modify data in-place. We prefer immutable data that cannot be accidentally changed, especially as a side-effect.

Lazy evaluation is the notion that we only evaluate as expression when we need to operate on it. This allows us to express and manipulate complex expressions.



The Functional Programming Paradigm.
https://towardsdatascience.com

Kotlin Support

Kotlin is a ***hybrid*** language that supports OO, FP and Imperative programming styles.

- It's up to you to decide the best approach for a particular problem, and you can mix paradigms.
- Think of FP as a toolbox of goodies that you can bring to bear on a problem.

1. **Mutation and side effects**

- Use `val` instead of `var`
- Avoid globals for carrying program state, when you can
- Favor functions that are cohesive, and free of side-effects ("*you mean it's that easy?*")

2. **First-class functions & higher-order functions**

- We'll spent most of this lecture on this topic!

Function Types

Functions as Types

Functions in Kotlin are "first-class citizens" of the language.

This means that we can define functions, assign them to variables, pass functions as arguments to other functions, or return functions! Functions are types in Kotlin, and we can use them anywhere we would expect to use a regular type.

Some definitions

- **Pure function**: a function that does not have any side effects.
- **Higher-order function**: a function that accepts another function as an argument, or returns a function.

Example: Barber Shop

Bert's Barber shop is creating a program to calculate the cost of a haircut, and they end up with 2 almost-identical functions (Leeds 2022).

```
fun calculateTotalWithFiveDollarDiscount(initialPrice: Double): Double {
  val priceAfterDiscount = initialPrice - 5.0
  val total = priceAfterDiscount * taxMultiplier
  return total
}

fun calculateTotalWithTenPercentDiscount(initialPrice: Double): Double {
  val priceAfterDiscount = initialPrice * 0.9
  val total = priceAfterDiscount * taxMultiplier
  return total
}
```

Functions are <u>identical</u> except for this code

If we could somehow pass in *that line of code as an argument*, then we could replace both with a single function that looks like this, where `applyDiscount()` represents the code that we would dynamically replace:

```
// applyDiscount = initialPrice * 0.9, or
// applyDiscount = initialPrice - 5.0

fun calculateTotal(initialPrice: Double, applyDiscount: ???): Double {
    val priceAfterDiscount = applyDiscount(initialPrice)
    val total = priceAfterDiscount * taxMultiplier
    return total
}
```

Assign a function to a variable

Here’s how we assign one of our functions to a variable.

```
fun discountFiveDollars(price: Double): Double = price - 5.0
val applyDiscount = ::discountFiveDollars
```

`applyDiscount` is now a reference to the `discountFiveDollars` function (note the `::` notation when we have a function on the RHS of an assignment). We can even call it.

```
val discountedPrice = applyDiscount(20.0) // Result is 15.0
```

The type of a function

So what is the type of our function?

```kotlin
// this is the original function signature
fun discountFiveDollars(price: Double): Double = price - 5.0
val applyDiscount = ::discountFiveDollars

// applyDiscount accepts a Double as an argument and returns a Double
// we use this format when specifying the type
val applyDiscount: (Double) -> Double

// we could use this format for other functions too
// note that this is now a val and not a fun
val discountFiveDollars: (Double) -> Double
```

Pass a function to a function

We can use this information to modify the earlier example:

```
fun discountFiveDollars(price: Double): Double = price - 5.0
fun discountTenPercent(price: Double): Double = price * 0.9
fun noDiscount(price: Double): Double = price
```

Function signatures match

```
fun calculateTotal(initialPrice: Double, applyDiscount: (Double) -> Double): Double {
    val priceAfterDiscount = applyDiscount(initialPrice)
    val total = priceAfterDiscount * taxMultiplier
    return total
}

val withFiveDollarsOff = calculateTotal(20.0, ::discountFiveDollars) // $16.35
val withTenPercentOff  = calculateTotal(20.0, ::discountTenPercent)  // $19.62
val fullPrice          = calculateTotal(20.0, ::noDiscount)          // $21.80
```

Return a function from a function

Instead of typing in the *name of the function* each time he calls `calculateTotal()`, Bert would like to just enter the *coupon code* from the bottom of the coupon that he receives from the customer.

To do this, he just needs a function that accepts the coupon code and returns the right discount function.

```
// accepts a String argument, and return a function
fun discountForCouponCode(couponCode: String): (Double) -> Double = when
(couponCode) {
    "FIVE_BUCKS" -> ::discountFiveDollars
    "TAKE_10"    -> ::discountTenPercent
    else         -> ::noDiscount
}
```

Intro to Lambdas

Function Literals

We can use this same notation to express the idea of a **function literal**, or a function as a value.

```
val applyDiscount: (Double) -> Double = { price: Double -> price - 5.0 }
val applyDiscount = { price: Double -> price - 5.0 } // type inferred
```

The code on the RHS of this expression is a function literal, which captures the body of this function. We also call this a **lambda**. A lambda is just an anonymous function, written in this form:

- the function is enclosed in curly braces { }
- the parameters are listed, followed by an arrow
- the body comes after the arrow

```
{ price: Double -> price - 5.0 }
```

A lambda expression.

The implicit “it”

Original forms:

```
val applyDiscount: (Double) -> Double = { price: Double -> price - 5.0 }
val applyDiscount = { price: Double -> price - 5.0 } // type inferred
```

In cases where there’s only a *single parameter* for a lambda, you can *omit the parameter name and the arrow*. When you do this, Kotlin will automatically make the name of the parameter `it`.

Shortened forms:

```
val applyDiscount: (Double) -> Double = { it - 5.0 }
// what about the type inferred version?
```

Lambdas as arguments

We can rewrite our earlier earlier example to use lambdas instead of function references:

```kotlin
// fun discountFiveDollars(price: Double): Double = price - 5.0
// fun discountTenPercent(price: Double): Double = price * 0.9
// fun noDiscount(price: Double): Double = price

fun calculateTotal(initialPrice: Double, applyDiscount: (Double) -> Double): Double {
    val priceAfterDiscount = applyDiscount(initialPrice)
    val total = priceAfterDiscount * taxMultiplier
    return total
}

val withFiveDollarsOff = calculateTotal(20.0, { price - 5.0 }) // $16.35
val withTenPercentOff  = calculateTotal(20.0, { price * 0.9 }) // $19.62
val fullPrice          = calculateTotal(20.0, { price })       // $21.80
```

In cases where function’s *last parameter* is a function type, you can move the lambda argument *outside* of the parentheses to the right, like this:

```kotlin
val withFiveDollarsOff = calculateTotal(20.0) { price -> price - 5.0 }
val withTenPercentOff  = calculateTotal(20.0) { price -> price * 0.9 }
val fullPrice          = calculateTotal(20.0) { price -> price }
```

This is meant to be read as **two arguments**: one inside the brackets, and the lambda as the second parameter.

Returning lambdas

We can easily modify our earlier function to return a lambda as well.

```
fun discountForCouponCode(couponCode: String): (Double) -> Double =
  when (couponCode) {
     "FIVE_BUCKS" -> { price -> price - 5.0 }
     "TAKE_10"    -> { price -> price * 0.9 }
     else         -> { price -> price }
 }
```

Collection Functions

Collections

Collection classes (e.g. List, Set, Map, Array) have built-in *pure functions* for working with their data. These functions frequently accept other function lambdas as arguments.

filter produces a new list of those elements that return true from a predicate function.

```
val list = (1..100).toList()
val filtered = list.filter { it % 5 == 0 } // 5 10 15 20 ... 100
```

map produces a new list that is the results of applying a function to every element that it contains.

```
val list = (1..100).toList()
val doubled = list.map { it * 2 } // 2 4 6 8 ... 200
```

reduce accumulates value starting with the first element and applying an operation to each element from left to right.

```
val strings = listOf("a", "b", "c", "d")
println(strings.reduce { acc, string -> acc + string }) // abcd
```

forEach calls a function for every element in the collection.

```
val fruits = listOf("advocado", "banana", "cantaloupe" )
fruits.forEach { print("$it ") }  // advocado banana cantaloupe
```

take returns a collection containing just the first n elements. **drop** returns a new collection with the first n elements removed.

```
val list = (1..50)
val first10 = list.take(10) // 1 2 3 ... 10
val last40 = list.drop(10)  // 11 12 13 ... 50
```

first and **last** return those respective elements. **slice** allows us to extract a range of elements into a new collection.

```
val list = (1..50)
val even = list.filter { it % 2 == 0 } // 2 4 6 8 10 ... 50
even.first()      // 2
even.last()       // 50
even.slice(1..3) // 4 6 8
```

Chaining Operations

We can chain operations together, so the return value of one function is acted on by the next function in the chain. e.g. map and filter a collection without needing to store the intermediate collection.

```
val list = (1..999999).toList()
val results = list
    .map { it * 2 }
    .take(10)
// [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

The operations are performed in top-down order: map, then take. In this case, it means that we're mapping the entire list and then discarding most of the resulting list with the take operation. This is really inefficient: filter your list first!

```
val veryLongList = listOf(0..9999999L).toList()
val results = veryLongList
    .take(10)
    .map { it * 2 }
// [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Resources

- John Guthrie ed. 2021. **Exploring Kotlin Functional Programming**. Manning Publications. ISBN 978-1617297090.
- Dave Leeds. 2022. **Dave Leeds on Kotlin**: Lambdas and Function References. https://typealias.com/start/kotlin-lambdas/
- Pierre-Yves Saumont. 2019. **The Joy of Kotlin**. Manning Publications. ISBN 978-1617295362.
- Venkat Subramaniam. 2019. **Programming Kotlin**. Pragmatic Bookshelf. ISBN 978-1680506358.
- Venkat Subramaniam . 2022. **Functional Programming in Kotlin**. Devoxx Poland. https://www.youtube.com/watch?v=emRPH2qeG48&t=1s