Kotlin Part 3: Functional Programming

CS 346: Application Development

What is 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.
- There are real benefits to this programming style:
  - Robustness
  - Expressivity
  - Clarity

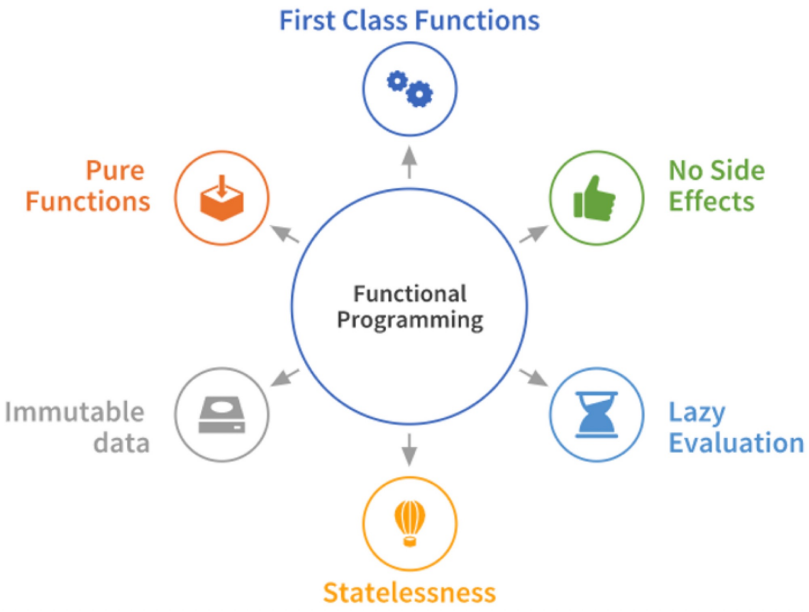
Don’t worry, we’re not bringing back Racket.

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

Functional Kotlin

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

How can we write Kotlin-style functional code?

- **Avoid unintended mutation and side effects**
  - Use `val` instead of `var`
  - Avoid globals for carrying program state, as much as possible
  - Favor pure functions that are free of side-effects i.e. avoid inline modification.
- **First-class functions & higher-order functions**
  - Explicitly functional expressions and constructs.
  - We'll spent most of this lecture on this topic!

Function Types

- Functions in Kotlin are "first-class citizens" of the language.
- ***Functions are types*** in Kotlin, and we can use them anywhere we would expect to use a regular type.
- This means that we can define functions, assign them to variables, pass functions as arguments to other functions, or return functions.
- Let's walk through some examples (with credit to Dave Leeds on Kotlin).

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.

```
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
}
```

Identical 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 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 invoke it.

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

The Type of a function

So what is the **type** of this function?

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


// we use this format when specifying the type
val applyDiscount: (Double) -> Double


// we could use this format for other functions too
val discountFiveDollars: (Double) -> Double
```

Pass a function to a function

```
fun discountFiveDollars(price: Double): Double = price - 5.0 // function signatures match
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, ::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 creates a function that accepts the coupon code and returns the correct 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
}
```

Function Literals (Lambdas)

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’

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.

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

Lambdas as Arguments

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

```
fun calculateTotal(initialPrice: Double, applyDiscount: (Double) -> Double): Double {
    val priceAfterDiscount = applyDiscount(initialPrice)
    val total = priceAfterDiscount * taxMultiplier
    return total
}
val withFiveDollarsOff = calculateTotal(20.0, { it - 5.0 }) // $16.35
val withTenPercentOff  = calculateTotal(20.0, { it * 0.9 }) // $19.62
val fullPrice          = calculateTotal(20.0, { it })       // $21.80
```

Trailing lambda

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:

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

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

This syntax, where the lambda function is placed outside of the brackets, is called a trailing lambda.

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 }
    }
```

Lambdas & Collections

Collection classes (e.g. List, Set, Map, Array) have built-in *pure functions* for working with their data.

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.

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

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

```
val strings = listOf("a", "b", "c", "d")
val str = 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
```