Lecture 1: Introduction to Object Oriented Programming with Kotlin 13-11-2023

Information about the lecturers

- Run by Robert Chatley
- rbc@imperial.ac.uk

Information about Kotlin

- To access the kotlin website, use kotlinlang.org for documentation etc.
- Use IntelliJ IDEA for Kotlin
- Kotlin is not whitespace sensitive
- Control + alt + L to reformat and align code automatically

Writing functions in Haskell and Kotlin

1. Here is an example of a successor function in Haskell vs Kotlin

The haskell version:

```haskell
successor :: Int -> Int
successor x = x + 1
```

The Kotlin version:

```kotlin
// Functions are defined with the 'fun' keyword
// They have a name, parameters in brackets, a return type, and a
definition
fun successor(x: Int): Int = x + 1
```

2. Here is another function example for finding if a value is even or not:

The Haskell version

```haskell
even :: Int -> Bool
even x = if x `mod` 2 == 0 then True else False
-- Nick definitely didn't approve of this
```

Here is the example in Kotlin

```kotlin
fun even(x: Int): Boolean = if (x%2==0) true else false
// The above version is the bad vesion that Nick hates
// Here is a better version below
fun even(x: Int): Boolean = x % 2 == 0
```

3. Another example in a difference function:

Haskell version:

```
difference :: Float -> Float -> Float
difference x y =
    | x > y = x - y
    | otherwise = y - x
```

Kotlin version:

```
// When in Haskell we would use guards, in Kotlin we use a 'when'
expression
fun difference(x: Double, y: Double): Double =
    when {
        x > y -> x - y
        else  -> y - x
    }
```

4. Here is an example of a function in Kotlin using when

```
fun (signum)(x: Int): Int
    when {
        x < 0 -> -1
        x > 0 -> 1
        else  -> 0
    }
```

5. Here is an example of a function that can be used to calculate the number of turns of a wheel to cover a distance from a start to an end point

The Haskell version of this function

```
turns :: Float -> Float -> Float -> Float
turns start end r = totalDistance / circumference
    where totalDistance = (end - start) * kmsToMs
          kmsToMs = 1000
          circumference = 2 * pi * r
```

The Kotlin version

```
// We'll need to import PI to be able to use PI on Kotlin, unlike Haskell
which incldues this from the beginning

import kotlin.math.PI

//...

fun turns(start: Double, end: Double, radius: Double): Double {
    val kmsToMs = 1000
    val totalDistance = (end - start) * kmToMs
    val circumference = 2 * PI * radius
    return totalDistance  / circumference
}
```

Calling functions in Kotlin

1. Running the difference function

```
difference(4.0, 6.0)
```

Practice problems in Kotlin - My answers

Try the following:

1. Write a function bigger() to determine which of two integers is the largest. Edit the code below to complete the function and then hit play to test your solution. You don't need to write a main() function here.

```
fun bigger(x: Int, y: Int): Int =
    when{
        x > y -> x
        else -> y
    }
```

2. Write a function biggestOfThree() to determine which of three integers is the largest:

```
fun biggestOfThree(x: Int, y: Int, z: Int): Int {
    if (x > y){
        if (x>z){
            return x
        }else{
            return z
        }
    }else{
        if (y > z){
            return y
        }else{
            return z
        }
    }
}
```

3. Write a recursive function fact() to calculate n factorial:

```
fun fact(n: Int): Int =
    when{
        n == 0 -> 1
        else -> fact(n-1) * n
    }
```

4. Write a recursive function fib() to calculate the nth Fibonacci number:

```
fun fib(n: Int): Int =
    when{
        n == 1 -> 1
        n == 0 -> 0
        else -> fib(n-1) + fib(n-2)
    }
```

Practice problems in Kotlin - Given answers

Try the following:

1. Write a function bigger() to determine which of two integers is the largest. Edit the code below to complete the function and then hit play to test your solution. You don't need to write a main() function here.

```
fun bigger(x: Int, y: Int): Int =
    when{
        x > y -> x
        else -> y
    }
```

2. Write a function biggestOfThree() to determine which of three integers is the largest:

```
fun biggestOfThree(x: Int, y: Int, z: Int): Int = bigger(x, bigger(y, z))
```

3. Write a recursive function fact() to calculate n factorial:

```
fun fact(n: Int): Int =
    when (n){
        0    -> 1
        else -> fact(n-1) * n
    }
```

4. Write a recursive function fib() to calculate the nth Fibonacci number:

```
fun fib(n: Int): Int =
    when (n) {
        0, 1 -> 0
        else -> fib(n-1) + fib(n-2)
    }
```

Data types

- We have covered functions, ints and Bools. Other common data types include:

Tuples

The following is an example of a function to calculate the length of the line segment between two points

```
//Using pairs, we can extract their contents using .first and .second

import kotlin.math.sqrt

//...

fun distanceBetween(a: Pair<Int, Int>, b: Paid<Int, Int>): Double {
    val dx = a.first - b.first
    val dy = a.second - b.second
    return (sqrt((dx * dx) + (dy * dy)).toDouble())
}
```

To use this function:

```
distanceBetween(Pair(2,3), Pair(10,12))
```

To construct pairs:

```
val a = Pair(2, 3)
val b = Pair(9, 11)
```

Strings

- Here are some methods on strings

```
val str = "Hello World"

str.uppercase() // Makes a new string that is uppercase. str is not
changed, this is just a new string that has been created that is uppercase

str.substring(2) // This takes the substring from the second index onwards.
This is `llo World`. This is inclusive of the substring

str.substring(2, 7) // This takes the substring from 2 to 7th index
inclusive. This is `llo W`
```

```
str.lowercase().startsWith("H") // This function checks if the lowercase
version starts with capital H.
```

- Here is a function defined using some methods on strings

```
fun h(s: String) =
when {
    s.startsWith("H) -> println("H)
    else             -> println("not H")j
}

h("Bye")
```

Lists

- To define a list, you can just write:

```
val xs = listOf(1,2,3,4,5,6)
```

- Kotlin will automatically figure out what the type of the elements of the list are
- If you want to manually declare this, write:

```
val xs: List<Int> = listOf(1,2,3,4,5,6)
```

- To create a list of Strings

```
val strings: List<String> = listOf("quick", "brown", "fox")
```

- Consider the function:

```
fun square(x: Int): Int = x * x
```

- To map a function onto the elements of the above list, run:

```
xs.map(::square)
//Use the double colon to pass a function that has already been defined as
a map
```

- You can define values that are functions:

```
val square2 = fun(x: Int): Int = x * x
// This is a function type, where the function type takes something and
returns another value
// This type is actually:
val square2: (Int) -> Int = fun(x: Int): Int = x * x
// You do not need to actually type the type as the compiler can figure it
out
```

- Now that we have defined this function as a value, we can do the following:

```
xs.map(square2)
//now we do not need a double colon
```

- To define an anonymous function

```
xs.map({x -> x * x})
// This the most common form of mapping
// The version of creating a value which is then used as a map is not as
common as using a lambda function
// We can actually write it as either of the two ways below, where the
lambda is the only or the last parameter of the method, we can write as
follows
xs.map {x -> x * x}
// or
xs.map () {x -> x * x}
```

- We can apply a lambda function twice

```
xs.map(squareFunc).map {y -> y + 1}
```

- You can define functions to apply functions to values

```
fun applyFuncTo(x: Int, f: (Int) -> Int): Int = f(x)
```

- Looking more specifically at HOFs and lists of strings

```
string.map { x-> s.uppercase()}
```

Lecture 2: Object Oriented Programming with Kotlin 14-11-2023

Constants in Kotlin

- We can define a constant as follows:

```
const val GOLDEN_RATIO = 1.618
```

- Values for constants are computed at compile time

Defining variables as lambda expressions and functions

- Functions can be defined with the same name as a variable
- If you have a variable that is equal to a lambda expression, and name a function with the same name as the variable, Kotlin will dynamically use the correct reference based on the situation

Defining functions polymorphically

- We can define a polymorphic function that composes two functions together as follows

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C = {x -> f(g(x))}

val l: (String) -> Int = {s: String -> s.length}
```

Getters and setters

- To access the elements of a list

```
val first = strs[0]
// The above gets hte first element
val second = strs[1]
// The above gets the second element
```

- You cannot wrap around like in python, i.e we can't do `strs[-1]`

Tuples and destructuring

- We can destructure a list as follows to extract the elements of a tuple

```
val p: Pair<Int, Int> = Pair(6,7)

val (f, s) = p
```

- The above will allow for you to access items of a tuple using destructuring
- If you do the same on a list:

```
var strs = listOf("quick", "brown", "fox")

val (f, s) = p
```

```
println(f) // This will return "quick"
println(s) // This will return "brown"
```

Higher Order Functions in Kotlin

- The map function will perform a lambda expression upon each element of a list
- The filter function will allow an element to remain in a list if the boolean output of a lambda expression is true

```
fun lengths(strs: List<String>): List<Int> {
    var k = strs.map {x: String -> x.length}
    return k
}
// This will return a list of lengths given a list
```

```
fun complements(nums: List<Int>): List<Pair<Int, Int>> {
    var k = nums.map {x -> Pair(x, 10-x)}
    return k
}
// This will return a list of pairs of complements given a list of ints
```

```
val pairs = listOf(Pair(1, 9), Pair(3, 4), Pair(5, 5))

fun matchingTotal(x: Int, pairs: List<Pair<Int, Int>>): List<Pair<Int,
Int>> {
    return pairs.filter {(m,y) -> m+y == x }
}
// This is an example of using filter to remove pairs from a list of pairs
who's sum of its elements does not meet a given integer
```