Lecture 3

COMP 3717- Mobile Dev with Android Tech

Functions

- Declared using the *fun* keyword

```
fun doSomething(){
    println("Hello world")
}
```

- The default access modifier is *public* which is same as variables

Functions (cont.)

- Functions have a default return type of Unit

```
fun greet() : Unit {
    println("Hello World")
}
```

- Java's version of this is *void*

Functions (cont.)

- Functions can also be declared outside of classes

```
fun main() {
    greet()
}

fun greet(){
    println("Hello World")
}
```

- This is handy when logic of a function does no clearly belong to a class

Arguments and Parameters

- We can create a function with parameters, then when we invoke it, we pass in the arguments

```
fun main() {
    greet( arg: "sponge")
}

fun greet(arg: String) {
    println("Hello $arg")
}
```

```
"C:\Program Files\Android\Android Stud
Hello sponge

Process finished with exit code 0
```

Named arguments

- Here we can change the order we pass in arguments

```
fun main() {
    greet( arg1: "sponge",  arg2: 3,  arg3: "star")
    greet(arg3="star", arg1="sponge", arg2=3) // using named arguments
}

fun greet(arg1: String, arg2: Int, arg3: String) {
    println("$arg1 has $arg2 friends, one of them is a $arg3")
}
```

```
"C:\Program Files\Android\Android Studio\jbr
sponge has 3 friends, one of them is a star
sponge has 3 friends, one of them is a star

Process finished with exit code 0
```

Default arguments

- If set a default value for a parameter, then we don't need to pass it in as an argument

```
fun main() {
    greet( arg1: "sponge",  arg2: 3,  arg3: "star")
    greet(arg3="whale", arg1="squirrel")

}

fun greet(arg1: String, arg2: Int = 3, arg3: String) {
    println("$arg1 has $arg2 friends, one of them is a $arg3")
}
```

```
"C:\Program Files\Android\Android Studio\jbr\bin
sponge has 3 friends, one of them is a star
squirrel has 3 friends, one of them is a whale

Process finished with exit code 0
```

Return statement

- To exit out of a function before it ends, we use the return statement

```
fun main() {

    greet( greeting: "Hello World",  hideFact: true)
    println("...end program")

}

fun greet(greeting:String, hideFact: Boolean){
    println(greeting)
    if (hideFact) return
    fact()
}

fun fact(){
    println("sponge has 3 friends and 1 of them is a squirrel")
}
```

```
"C:\Program Files\Android\Android Stu
Hello World
...end program

Process finished with exit code 0
```

Return value from function

- A function can also return a value
  - Once it returns a value it will exit the function

```
fun main() {

    val i = 30
    println(double(i))
}

fun double(arg: Int) : Int{
    return arg * 2
}
```

```
"C:\Program Files\Android\Android St
60

Process finished with exit code 0
```

Single expression function

- A function can be reduced down to a single expression if it returns a single statement

- Notice we can also omit the return type

```
fun main() {

    val i = 30
    println(double(i))
}

fun double(arg: Int) = arg * 2

//fun double(arg: Int) : Int{
//    return arg * 2
//}
```

Return multiple values

- We can return a Pair of values from a function

```
fun twoValues() : Pair<String, Int>{
    return "sponge" to 3
    //return Pair("sponge", 3)
}
```

- Or a Triple

```
fun threeValues() : Triple<String, Int, String> =
    Triple(
        first: "sponge",
        second: 3,
        third: "star"
    )
```

Return multiple values (cont.)

- You can also deconstruct a Pair and Triple

```
val(species, numFriends, friend) = threeValues()

println("$species has $numFriends friends, one is $friend")
```

```
"C:\Program Files\Android\Android St
sponge has 3 friends, one is star

Process finished with exit code 0
```

Maps

- To create a list of key-value pairs we can use a Map

```
val map = mapOf(
    "key1" to "value1",
    "key2" to "value2",
    "key3" to "value3"
)

println(map["key1"])
println(map.keys)
println(map.values)
```

```
"C:\Program Files\Android\Android Studio\j
value1
[key1, key2, key3]
[value1, value2, value3]

Process finished with exit code 0
```

Maps (cont.)

- Maps can't have duplicate keys, and will overwrite the key with the new value

```
val map = mapOf(
    "key1" to "value1",
    "key2" to "value2",
    "key3" to "value3",
    "key3" to "value4",
    "key4" to "value1"
)

println(map.keys)
println(map.values)
```

```
"C:\Program Files\Android\Android Studio
[key1, key2, key3, key4]
[value1, value2, value4, value1]

Process finished with exit code 0
```

- Maps can have duplicate values, if the key is different

Maps (cont.)

- To change the contents of the Map we need to use a *MutableMap*

```
val map = mutableMapOf(
    "key1" to "value1",
    "key2" to "value2",
    "key3" to "value3"
)

map["key1"] = "value1x"
map["key4"] = "value4"

println(map)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\java.e
{key1=value1x, key2=value2, key3=value3, key4=value4}

Process finished with exit code 0
```

HashMap vs MutableMap

- Difference between HashMap and MutableMap
  - MutableMap keeps entries in *order they were inserted*
  - With a HashMap, the *order of entries aren't guaranteed*
- Use a HashMap over a MutableMap if order doesn't matter

HashMap vs MutableMap (cont.)

```
val map = mutableMapOf("sponge" to "value1", "star" to "value2", "crab" to "value3")
val hashmap = hashMapOf("sponge" to "value1", "star" to "value2", "crab" to "value3")

map["squirrel"] = "value4"
hashmap["squirrel"] = "value4"

println(map)
println(hashmap)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\java.exe" ...
{sponge=value1, star=value2, crab=value3, squirrel=value4}
{sponge=value1, squirrel=value4, star=value2, crab=value3}

Process finished with exit code 0
```

Sets

- A *Set* is similar to a List but it can only have unique elements

```
val list = listOf("sponge", "star", "crab", "sponge")
val set = setOf("sponge", "star", "crab", "sponge")

println(list)
println(set)
```

```
"C:\Program Files\Android\Android Studi
[sponge, star, crab, sponge]
[sponge, star, crab]

Process finished with exit code 0
```

Sets

- To access the element at an index you can use *elementAt*

```
val set = setOf("sponge", "star", "crab")
println(set.elementAt( index: 0))
```

```
"C:\Program Files\Android\Android S
sponge

Process finished with exit code 0
```

Sets (cont.)

- To add/remove elements in a *Set* we can use *MutableSet*

```
val set = mutableSetOf("sponge", "star", "crab")
set.add("whale")
set.remove( element: "sponge")
println(set)
```

```
"C:\Program Files\Android\Android Studio\
[star, crab, whale]

Process finished with exit code 0
```

Sets vs Lists

- You cannot access the index of a *Set* and change it's contents

```
val list = mutableListOf("sponge", "star", "crab")
val set = mutableSetOf("sponge", "star", "crab")

list[0] = "squirrel"
set[0] = "squirrel"
```

- Use a *Set* over a *List* if you are working with unique elements

HashSet vs MutableSet

- Difference between HashSet and MutableSet
  - MutableSet keeps entries in *order they were inserted*
  - With a HashSet, the *order of entries aren't guaranteed*
- Use a HashSet over a MutableSet if order doesn't matter

HashSet vs MutableSet

```
val set = mutableSetOf("sponge", "star", "crab")
val hashset = hashSetOf("sponge", "star", "crab")

set.add("whale")
hashset.add("whale")

println(set)
println(hashset)
```

```
"C:\Program Files\Android\Android Stu
[sponge, star, crab, whale]
[sponge, star, whale, crab]

Process finished with exit code 0
```

Functions can be assigned to variables

- Kotlin functions are first class, which means they can be assigned to variables (as seen below), and they are of higher order

```
fun main() {

    val fact1 = fact( name: "sponge",  friends: 3,  friend: "star")
    println(fact1)
```

```
fun fact(name: String, friends: Int, friend:String) : String{
    return "$name has $friends friends, one of them is a $friend"
}
```

Anonymous functions

- Kotlin allows us to create anonymous functions
- We use the fun keyword without a function name

```
fun main() {

    val factAnonymous = fun (name:String, friends:Int, friend:String) : String{
        return "$name has $friends friends, one of them is a $friend"
    }

    val fact2 = factAnonymous("squirrel", 4, "whale")
    println(fact2)
```

Function literals

- Function literals (aka. lambdas) have a slightly different syntax than anonymous functions

```
fun main() {

    val fact1: () -> Unit = { println("sponge has 3 friends and one of them is a star") }
    fact1()
```

- This lambda has no arguments and returns Unit

- When there are no arguments, the code body just provides the return type, which in this case is a Unit

Lambdas (cont.)

- If our lambda has one argument , we use the *it* keyword to refer to it in the expression
  - it: implicit name of a single parameter

```
fun main() {

    val fact1: (String) -> Unit = { it: String
        println(it)
    }

    fact1("sponge has 3 friends and one of them is a star")
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\
sponge has 3 friends and one of them is a star

Process finished with exit code 0
```

Lambdas (cont.)

- Here is another example using one argument

```
fun main() {

    val fact1: (Int) -> String = { it: Int
        "sponge has $it friends and one of them is a star"
    }

    println(fact1(3))
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\
sponge has 3 friends and one of them is a star

Process finished with exit code 0
```

Lambdas (cont.)

- When a lambda has multiple arguments we define these in the body { }

```
val fact1: (Int, String) -> Unit = { friends:Int, name:String ->
    println("$name has $friends friends and one of them is a star")
}

fact1(3, "sponge")
```

- Whatever this function returns, comes after the -> in the body { }

Lambdas (cont.)

- Here is another example using multiple arguments

```
val fact: (String, Int, String) -> String = { name:String, friends:Int, friend:String ->
    "$name has $friends friends and one of them is a $friend"
}

println(fact("sponge", 3, "star"))
```

```
"C:\Program Files\Android\Android Studio\jbr\bin
sponge has 3 friends and one of them is a star

Process finished with exit code 0
```

Lambdas (cont.)

- Just like when declaring other variables, lambdas can also be declared using type inference

```
val fact = { name:String, friends:Int, friend:String ->
    "$name has $friends friends and one of them is a $friend"
}

println(fact("sponge", 3, "star"))
```


SEE THE TIME ON THE CLOCK?
IT'S BREAK TIME!

Functions as parameters

- Kotlin uses *higher order functions* which means
  - Functions can be parameters
  - Functions can be returned from other functions

- Here we are setting a function with a return type of Unit as a parameter
  - The syntax uses a function literal (aka. lambda)

```
fun greet(fact: () -> Unit){
    println("Hello World")
    fact()
}
```

Functions as parameters (cont.)

- Just like other parameters we can provide a default value

```
fun greet(fact: () -> Unit = {}){
    println("Hello World")
    fact()
}
```

Functions as parameters (cont.)

- We could use a named argument to invoke the function argument

```
fun main() {

greet(fact={
    println("sponge has 3 friends, one of them is a star")
})
```

```
"C:\Program Files\Android\Android Studio\jbr\
Hello World
sponge has 3 friends, one of them is a star

Process finished with exit code 0
```

Functions as parameters (cont.)

- Or if a function literal is the last parameter, then it can be invoked at the end of the whole function (aka. trailing lambda)

```
fun main() {

    greet( arg: "Hello World") { name:String, friends:Int ->
        println("$name has $friends friends, one of them is a star")
    }

}

fun greet(arg:String, fact: (String, Int) -> Unit){
    println(arg)
    fact("sponge", 3)
}
```

Functions as parameters (cont.)

- You could also assign the whole expression to a variable and pass that

```
fun main() {

    val fact = { name:String, friends:Int ->
        println("$name has $friends friends, one of them is a star")
    }

    greet( arg: "Hello World", fact)

}

fun greet(arg:String, fact: (String, Int) -> Unit){
    println(arg)
    fact("sponge", 3)
}
```

Double colon :: operator

- Its common to have a regular prebuilt function that you want to pass around instead of a lambda

- As long as the parameters match, the double colon :: operator allows us to pass a function reference

Double colon :: operator (cont.)

- Here we can pass a function reference because the function parameters both match

```
fun main() {

    greet( arg: "Hello World", ::fact)
}

fun greet(arg:String, fact: (String) -> String){
    println(arg)
    println(fact("sponge"))
}

fun fact(name:String) : String{
    return "$name has 3 friends, one of them is a star"
}
```

```
"C:\Program Files\Android\Android Studio\jbr\
Hello World
sponge has 3 friends, one of them is a star

Process finished with exit code 0
```

Double colon :: operator (cont.)

- If the function parameters do not match, than you need to provide default values

```
fun main() {

    greet( arg: "Hello World", ::fact)
}

fun greet(arg:String, fact: (String) -> String){
    println(arg)
    println(fact("sponge"))
}

fun fact(name:String, friends:Int = 3) : String{
    return "$name has $friends friends, one of them is a star"
}
```

```
"C:\Program Files\Android\Android Studio\jbr\
Hello World
sponge has 3 friends, one of them is a star

Process finished with exit code 0
```

Class Activity 1

- Recreate this code using a lambda function instead of a function reference

```
fun main() {

    val operation1 = ::process
    println(operation1( type: "Bits",  amount: 40))
}

fun process(type:String, amount:Int) : String{
    return "...Processing $amount $type"
}
```

Class Activity 1 Answer

```
fun main() {

    val operation1 = {type:String, amount:Int -> String
        "...Processing $amount $type" ^lambda
    }
    println(operation1("Bits", 40))

}
```

Class Activity 2

- Looking at how *greet* is invoked:

```
fun main() {

    greet( arg1: "Class Activity 2"){ it: String
        "$it World"
    }
}
```

- And also the output in the console:

```
"C:\Program Files\Android\Android St
Class Activity 2: Hello World

Process finished with exit code 0
```

- Create the *greet* function

Class Activity 2 Answer

```
fun greet(arg1:String, arg2:(String)->String){
    println("$arg1: ${arg2("Hello")}")
}
```