Lecture 5

COMP 3717- Mobile Dev with Android Tech

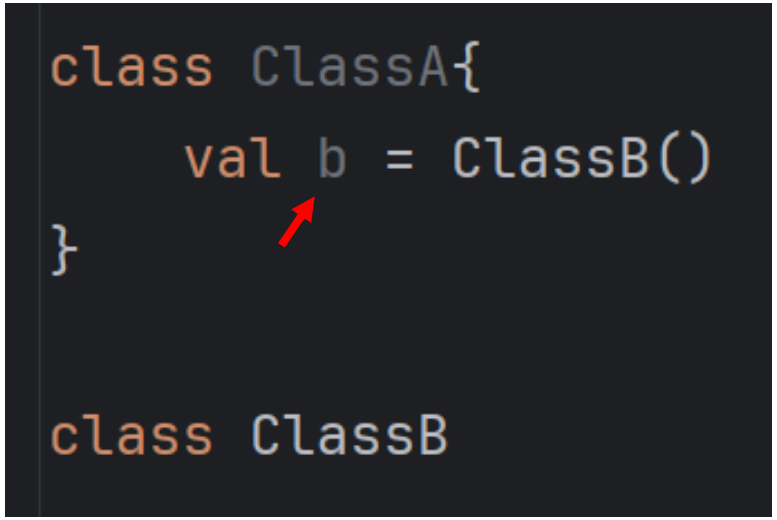
Association

- Association is a *has-a* relationship between classes

- There are two ways to achieve association
  - Composition
  - Aggregation

Association (cont.)

- Composition is when an objects lifecycle is determined by its parent



- When a *ClassA* object is created/destroyed, so is *b*

Association (cont.)

- Aggregation is when the objects lifecycle is not determined by its parent

- *b* is created/destroyed outside of *ClassA*

Association vs Inheritance

- Inheritance has it's use cases, but often a *has-a* relationship is better

- Consider *POSSystem* and *Restaurant* below

```
interface POSSystem{
    fun processOrder()
    fun calculateFoodInventory()
}
```

```
open class Restaurant : POSSystem{
    override fun processOrder() {
        println("Processing Order...")
    }
    override fun calculateFoodInventory() {
        println("Calculating food inventory...")
    }
}
```

Association vs Inheritance (cont.)

- First off, we can achieve the same functionality using either relationship

- is-a

```
class KrustyKrab : Restaurant()
```

- has-a with delegation (*r* is the delegate)

```
class KrustyKrab{
    private val r = Restaurant()
    fun processOrder() = r.processOrder()
    fun calculateFoodInventory() = r.calculateFoodInventory()
}
```

Delegation

- Delegation is a way of achieving *association* (has-a) between two objects rather than *inheriting* (is-a)

- Delegation is a design pattern that Kotlin supports natively with interfaces
  - Uses the **by** keyword

- Delegation means delegating responsibilities to another object

Delegation (cont.)

- If we use *POSSystem* as the delegate instead of *Restaurant*

```
class KrustyKrab(private val posSystem: POSSystem){
    fun processOrder() = posSystem.processOrder()
    fun calculateFoodInventory() = posSystem.calculateFoodInventory()
}
```

- The delegate becomes more flexible
  - This is harder to achieve with inheritance

Delegation (cont.)

- We can achieve the same logic by implementing *POSSystem* and overriding its members

```
class KrustyKrab(private val posSystem: POSSystem) : POSSystem{
    override fun processOrder() {
        posSystem.processOrder()
    }

    override fun calculateFoodInventory() {
        posSystem.calculateFoodInventory()
    }
}
```

Delegation (cont.)

- Furthermore, Kotlin supports interface delegation natively by reducing a lot of the boilerplate code

```
class KrustyKrab(posSystem: POSSystem) : POSSystem by posSystem
```

- The *by* keyword means *provided by* the delegate

Delegation (cont.)

- Summary
  - With inheritance, we are limited to the parent class type
  - With delegation, the delegate can be swapped easily making our class more flexible and reusable
  - The *by* keyword in Kotlin is used to delegate responsibility to something else

Delegation (cont.)

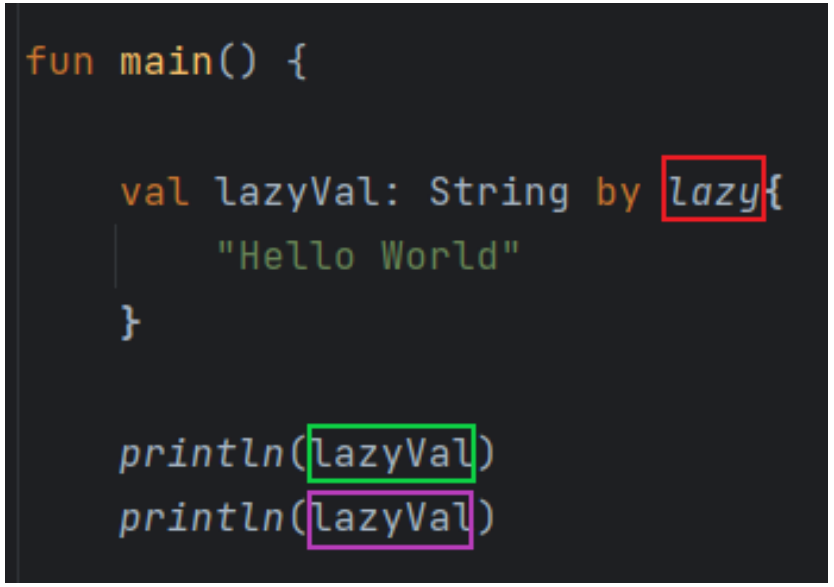
- Kotlin also provides some standard delegates
  - Lazy
  - Observable
  - Vetoable

Lazy delegate

- All properties can use the lazy delegate
- Initializing a property as lazy means it will only be initialized when it is first used
- Since it is only initialized the first time is it used, by nature it can only be declared as *val*

Lazy delegate(cont.)

- When we make a property lazy, we delegate its getter to the *lazy* delegate



- The first call to get will execute the lambda and remember it's result
- Any call to get afterwards will return the remembered result

Lazy delegate(cont.)

- Use the lazy delegate if you need
  - a read-only property, where you want to delay or avoid its initialization

- Let's say we have a regular class that has-a property that performs a heavy operation

```
fun main() {
    RegularClass()
}

class HeavyClass{
    init{
        println("Some heavy processing...")
    }
}

class RegularClass{
    val heavy = HeavyClass()
}
```

```
"C:\Program Files\Android\Android Stu
Some heavy processing...

Process finished with exit code 0
```

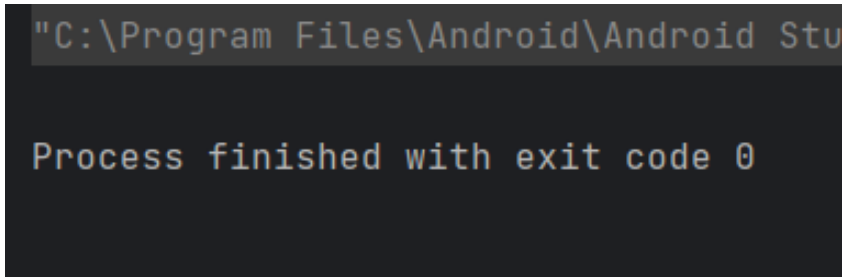
Lazy delegate(cont.)

- By making the heavy property lazy, we avoid its initialization

```
fun main() {
    RegularClass()
}

class HeavyClass{
    init{
        println("Some heavy processing...")
    }
}

class RegularClass{
    //val heavy = HeavyClass()
    val heavy by lazy {
        HeavyClass()
    }
}
```



Observable delegate

- To observe the changes to a property you can use the observable delegate

```
class Sponge{
    var name: String by Delegates.observable( initialValue: "Bob"){ _, oldValue, newValue ->
        println("old value: $oldValue")
        println("new value: $newValue")
    }
}
```

- We delegate the property's setter to the *observable* delegate
  - Provides the old and new value whenever we set our property

Observable delegate (cont.)

- Each time we set *name*, the observable delegate is invoked

```
fun main() {

    val sponge = Sponge()
    sponge.name = "Spongebob"
    sponge.name = "Mr. SquarePants"
}
```

```
"C:\Program Files\Android\Android Stu
old value: Bob
new value: Spongebob
old value: Spongebob
new value: Mr. SquarePants

Process finished with exit code 0
```

Vetoable delegate

- To veto the changes to a property, we use the vetoable delegate

```
class Sponge{
    var friends:Int by Delegates.vetoable( initialValue: 3){_, oldValue, newValue ->
        println("old value: $oldValue")
        println("new value: $newValue")
        newValue >= 0 ^vetoable
    }
}
```

- We delegate the property's setter to a *vetoable* delegate
  - Provides the old and new value (like observable)
  - If the lambda returns false, then the changes will be vetoed

Vetoable delegate (cont.)

- When we try to set *friends* to a vetoable condition, the value doesn't change

```
fun main() {

    val sponge = Sponge()
    sponge.friends = -4
    println(sponge.friends)
    sponge.friends = 4
    println(sponge.friends)
}
```

```
"C:\Program Files\Android\Android St
old value: 3
new value: -4
3
old value: 3
new value: 4
4

Process finished with exit code 0
```

Extension Functions

- Often, we are working with classes that we don't own

```
fun main() {

    val str: String = "free krabby patties"
}
```

- E.g., we didn't create the *String* class, it comes from the Kotlin standard library

Extension Functions (cont.)

- We might need some custom functionality from that class
- Usually in this case we would create our own function

```
fun myGetAllWords(str:String): List<String>{
    return str.split( ...delimiters: " ")
}
```

- It would be better though if we could call this function as part of the original class

Extension Functions (cont.)

- Below is how we can transform that logic into an extension function

```
fun String.getAllWords() : List<String>{
    return this.split( ...delimiters: " ")
}
```

- The class you want to extend
  - aka. the receiver

- The instance of the class you are extending

Extension Functions (cont.)

- You can also drop *this*

- In summary, extension functions
  - allow us to add functionality to classes we don't own; and
  - call these new functions as if they were part of the original class

Extension Properties

- We can create extension properties as well
  - The *receiver is a String*
  - The instance

Lambda with Receiver

- You can also create *extension function literals*
  - aka. Function literal with a receiver; or
  - *lambda with a receiver*

```
val reverseDigits: Int.() -> Int = { this: Int
    toString().reversed().toInt()
}
```

Lambda with Receiver

- Providing the reciever can be done in two ways
  - Standard

```
println(145.reverseDigits())
```

  - More explicit

```
println(reverseDigits(145))
```

Provides the reciever as p1 (parameter 1) which is what is happening behind the scenes when working with a *lambda with reciever*

Lambda with Receiver (cont.)

- Furthermore, they can help give us scope to object members

```
//class we don't have access too
class Sponge{
    val name = "Spongebob"
}

fun Sponge.fact(action: Sponge.()->Unit){
    this.action()
}
```

```
fun main() {
    val sponge = Sponge()
    sponge.fact {
        println("$name lives in a pineapple")
    }
}
```

Generics

- Generics allows us to define functions or classes that can work with different data types (Int, String, Double, etc)

- This is useful because we only have to write one piece of code for multiple types

Generics

- Let's say we want to create some custom sorting behaviour for a list

- We want to be able to find an element in our list and sort it to the first index
  1.) First, we find the index of the element we want to sort
  2.) Remove it from the list
  3.) Add it to the first index

```
val indexOfElement = list.indexOf(element)
list.removeAt(indexOfElement)
list.add( index: 0, element)
```

Generics (cont.)

- Let's add this code to a class so we can easily reuse it
- We will pass the list into the constructor and the element into our sort function
- *indexOf* returns -1 if the element doesn't exist
  - We will just return the original list in this case

```
class IntListUtils(private val data: List<Int>) {

    fun sortElementFirst(element: Int): List<Int> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

Generics (cont.)

- This code works well, it finds the first given element and sorts it to the front of the list

```
fun main() {

    val list = listOf(1, 2, 3, 4, 5, 6)
    val customSortedList = IntListUtils(list).sortElementFirst( element: 3)
    println(customSortedList)

}
```

```
"C:\Program Files\Android\Android St
[3, 1, 2, 4, 5, 6]

Process finished with exit code 0
```

Generics (cont.)

- The only problem is that our code only works for integers
  - What if we want to use this code for other types?

- When we make a class use generics, we need to add <T> to the end of the name

```
class ListUtils<T>(private val data: List<Int>) {
```

Generics (cont.)

- You can use any letter for the generic type

- Some commonly used type parameter names are
  - T - Type
  - K - Key
  - V - Value
  - S,U,V etc. - 2nd, 3rd, 4th types

Generics (cont.)

- Once you have created a generic type for your class
- All other classes, objects and functions in that class can use that generic type as parameters

Generics (cont.)

- Our class is using generics now and can handle any type of list

```
fun main() {

    val list = listOf("1", "2", "3", "4", "5", "6")
    val customSortedList = ListUtils(list).sortElementFirst( element: "3")
    println(customSortedList)
```

```
"C:\Program Files\Android\Android Stu
[3, 1, 2, 4, 5, 6]

Process finished with exit code 0
```

Generics (cont.)

- We can also just make a specific function generic, rather than a whole class
- The *<T>* is added to the front of the function name

```
class ListUtils {

    fun <T>sortElementFirst(data: List<T>, element: T): List<T> {
        val list = data.toMutableList()
        val indexOfElement = list.indexOf(element)

        return if (indexOfElement != -1) {
            list.removeAt(indexOfElement)
            list.add( index: 0, element)
            return list
        } else {
            println("element doesn't exist")
            data
        }
    }
}
```

```
fun main() {

    val list = listOf("1", "2", "3", "4", "5", "6")
    val customSortedList = ListUtils().sortElementFirst(list,  element: "3")
    println(customSortedList)
```

Generics (cont.)

- Why can't we just use *Any*?

```
fun sortElementFirst(data: List<Any>, element: Any): List<Any> {
    val list = data.toMutableList()
```

- Generics provides type inference, so it will detect the specific type

- *Any* does not have any type inference making it harder to work with

Comparing objects

- Sometimes you might want to compare objects rather than strings or primitive types

```
class Species(private val name:String)

fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")

    println(sponge1 == sponge2)
```

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

Process finished with exit code 0
```

Comparing objects (cont.)

- In the previous example you might wonder why the result is false
- Object variables, or any object that is an instance of a class, *store their values as references* (aka. a location in memory)
- Primitive types *store their values with the data directly*
- When you compare two variables using the equality operator (==), we compare their values

Pass by value

- Like Java, Kotlin passes its variables by value to functions (aka. *Pass-by-value*)

```
fun doSomething(s:Species){
    s.name = "Mr. SquarePants"
}

fun main() {

    val sponge = Species( name: "Bob")
    doSomething(sponge)
    println(sponge.name)
```

```
"C:\Program Files\Android\Android Studio\j
Mr. SquarePants

Process finished with exit code 0
```

- We pass in the value (*a copy* of the memory location) of *sponge* then alter the sponge object's data


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

Data Class

- In Java, it is common to create a data class (POJO)
  - A simple class mainly used to hold data
- It is also common in Java for the POJO to
  - Implement getters/setters for each data member
  - Override the superclass's (*Object*) *toString*, *equals*, and *hashcode* functions
    - Kotlin uses Any not Object
- Doing all this helps us manage and transfer our data

Data Class (cont.)

- For example, we might want all our data to be represented in a String

```
class Species(
    val name:String,
    val height:Int,
    val occupation:String
)

fun main() {

    val sponge = Species( name: "Bob",  height: 4,  occupation: "Cook")
    println(sponge)
```

```
"C:\Program Files\Android\Android Studio
com.bcit.lecture5.Species@6cd8737

Process finished with exit code 0
```

- But you can see the output isn't very helpful

Data Class (cont.)

- We also saw earlier that it is common to compare two objects

```
class Species(private val name:String)

fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")

    println(sponge1 == sponge2)
```

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

Process finished with exit code 0
```

- If we only care about comparing the data in the two objects, then we would want this to print true

Data Class (cont.)

- One way to get around this is to override those superclass functions
  - toString, equals, hashCode
- We would need to write similar boilerplate code for all our POJO's in Java

```
class Species(val name:String){

    override fun toString(): String {
        return "Species(name=$name)"
    }

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other == null) return false
        if (other !is Species) return false
        return name == other.name
    }

    override fun hashCode(): Int {
        return name.hashCode()
    }
}
```

```
fun main() {

    val sponge1 = Species( name: "Bob")
    val sponge2 = Species( name: "Bob")
    println(sponge1)
    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android S
Species(name=Bob)
true

Process finished with exit code 0
```

Data Class (cont.)

- In Kotlin, all we must do is make the class a *data class* and we can achieve the same thing

```
data class Species(val name:String, val height:Int, val occupation:String)

fun main() {

    val sponge1 = Species( name: "Bob",  height: 4,  occupation: "Cook")
    val sponge2 = Species( name: "Bob",  height: 4,  occupation: "Cook")
    println(sponge1)
    println(sponge1 == sponge2)
```

```
"C:\Program Files\Android\Android Studio\jbr\b
Species(name=Bob, height=4, occupation=Cook)
true

Process finished with exit code 0
```

Data Class (cont.)

- Can't be *open* or *abstract*
- Can't have a blank constructor (needs at least one parameter)
- Can be deconstructed

```
data class Species(val name:String, val height:Int, val occupation:String)

fun main() {

    val sponge = Species( name: "Bob",  height: 4,  occupation: "cook")
    val (name, height, occupation) = sponge
    println("$name is a $occupation who is $height feet tall")
```

```
"C:\Program Files\Android\Android Studi
Bob is a cook who is 4 feet tall

Process finished with exit code 0
```

Scope functions

- Scope functions are functions that execute a block of code within the context of an object

- The give *temporary scope* to an object where *specific operations* can be applied

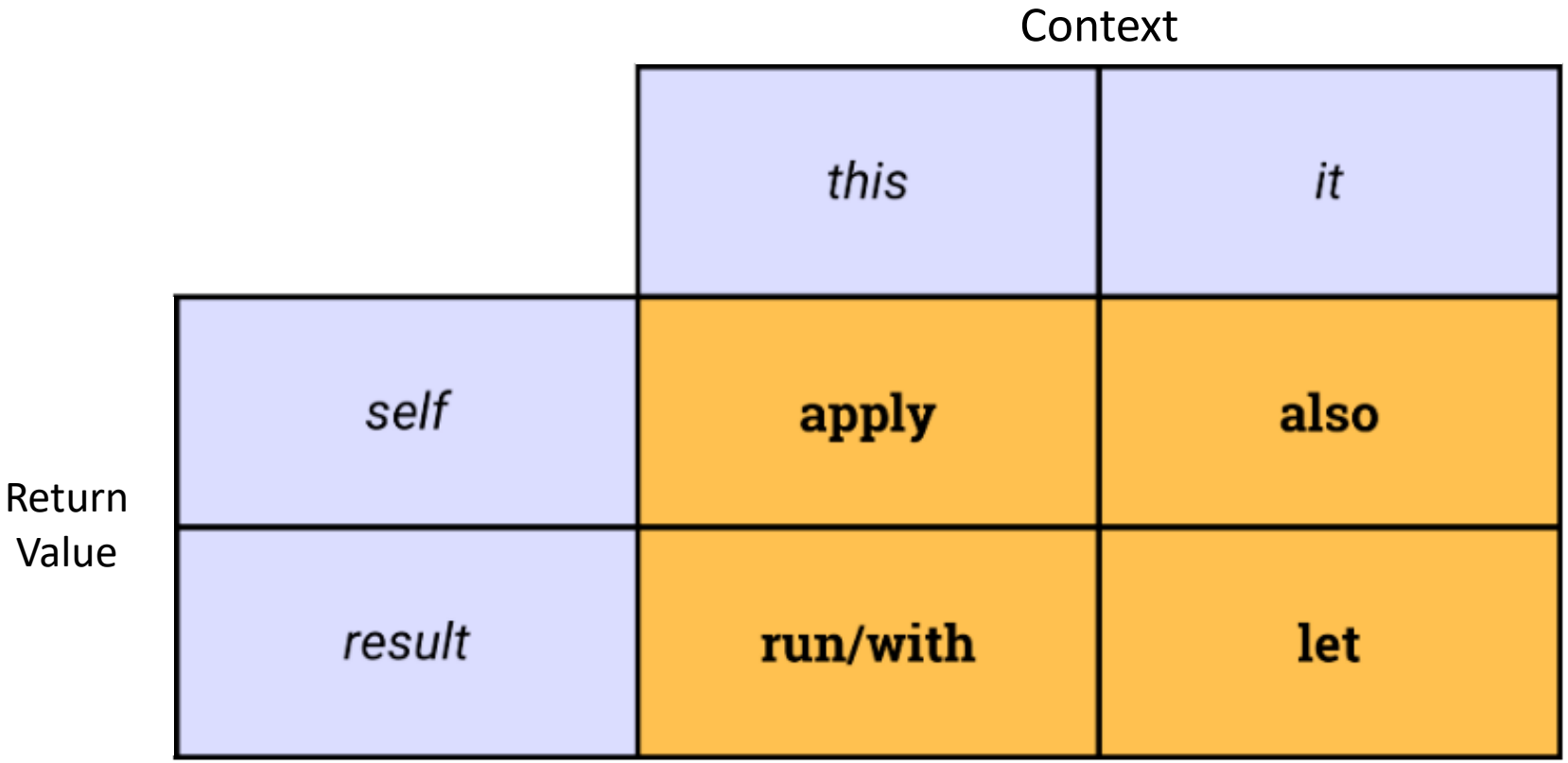
- Scope functions make our code more concise and readable

Scope functions (cont.)

- There are five scope functions provided in the Kotlin standard library
  - *let*, *run*, *with*, *apply*, *also*

- We can differentiate these scope functions in two ways
  - Context: *it* or *this*
  - Return Value: *lambda result* or *itself (context object)*

Scope functions (cont.)



Scope functions (cont.)

- Consider this class below

```
class Car(
    private var make:String? = null,
    private var color:String? = null,
    private var kilometers:Int
){
    fun drive(k:Int){
        kilometers += k
        println("The car drove $k kilometers")
    }

    fun paint(c:String){
        color = c
        println("Car has been panted $c")
    }

    override fun toString(): String {
        return "Car(make=$make, color=$color, kilometers=$kilometers)"
    }
}
```

- We will use this to explain the scope functions

Scope function let

- *let*
  - Context: *it*
  - Return Value: *lambda result*

- Here we create a new Car object with *.let{}* at the end
  - The *scope* is defined in the curly braces

```
val car = Car( make: "ford",  color: "red",  kilometers: 50).let{ it: Car

}
```

Scope function let (cont.)

- Inside the scope block, *it* refers to the Car object itself

- Since let *returns the lambda result*, the last line in the block is what it returned

```
val car = Car( make: "ford",  color: "red",  kilometers: 50).let{ it: Car
    it.drive( k: 50)
    it.paint( c: "Blue")
    it ^let
}
println(car)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\
The car drove 50 kilometers
Car has been panted Blue
Car(make=ford, color=Blue, kilometers=100)

Process finished with exit code 0
```

Scope function let (cont.)

- The *let* function is often used with null safety

```
var car1 = Car( make: "ford",  color: "red",  kilometers: 50)
val car2: Car? = null

car2?.let { car1 = it }

println(car1)
```

- Since *car2* is null, the let function won't be applied

Scope function run

- *run*
  - Context: *this*
  - Return Value: *lambda result*

- Very similar to *let* but we don't use *it*, we use *this*

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).run { this: Car
    this.drive( k: 60)
    this.paint( c: "Grey")
    this ^run
}
println(car1)
```

Scope function run (cont.)

- When the context is using *this*, we can drop *this* completely

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).run { this: Car
    drive( k: 60)
    paint( c: "Grey")
    this ^run
}
println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr
The car drove 60 kilometers
Car has been panted Grey
Car(make=ford, color=Grey, kilometers=110)

Process finished with exit code 0
```

Scope function with

- *with*
  - Context: *this*
  - Return Value: *lambda result*

- *with* is similar to *run* but the syntax is slightly different

- We can also omit returning *this* if we want

```
fun main() {

    val car1 = Car( make: "ford",  color: "red",  kilometers: 50)

    with(car1){ this: Car
        drive( k: 80)
        paint( c: "Blue")
        drive( k: 40)
    }

    println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\
The car drove 80 kilometers
Car has been panted Blue
The car drove 40 kilometers
Car(make=ford, color=Blue, kilometers=170)

Process finished with exit code 0
```

Scope function apply

- *apply*
  - Context: *this*
  - Return Value: *itself (context object)*

- Since we return the context object, we don't need to return anything in the block

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).apply { this: Car
    drive( k: 20)
    paint( c: "Black")
}

println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin
The car drove 20 kilometers
Car has been panted Black
Car(make=ford, color=Black, kilometers=70)

Process finished with exit code 0
```

Scope function also

- *also*
  - Context: *it*
  - Return Value: *itself (context object)*

- Similar to *apply* but we use *it* instead of *this*

```
val car1 = Car( make: "ford",  color: "red",  kilometers: 50).also { it: Car
    it.drive( k: 30)
    it.paint( c: "White")
    it.drive( k: 60)
}

println(car1)
```

```
"C:\Program Files\Android\Android Studio\jbr\bin\j
The car drove 30 kilometers
Car has been panted White
The car drove 60 kilometers
Car(make=ford, color=White, kilometers=140)

Process finished with exit code 0
```

Class Activity 1

- Given the Dog class, refactor the code in the main

```
class Dog{
    var name:String = "fluffy"
    var toys:MutableList<String> = mutableListOf()
    override fun toString(): String {
        return "Dog(name=$name, toys=$toys)"
    }
}
```

- Your code must use two different scope functions

```
fun main() {

    val dog1 = Dog()
    val dog2 = Dog()

    dog1.name = "sparky"
    dog2.toys.add("ball")
    dog2.toys.add("stick")
    dog1.toys = dog2.toys

    println(dog1)
```

Class Activity 1 Answer

```
fun main() {

    val dog1 = Dog()
    val dog2 = Dog()

    with(dog1){ this: Dog
        name = "sparky"
        toys = dog2.toys.also { it: MutableList<String>
            it.add("ball")
            it.add("stick")
        }
    }

    println(dog1)
```