# Developing Android Apps with Kotlin

NTG Clarity Networks Inc.

# Classes

- Syntax
  - `class ClassName [class-header][{[class-body]}]`

```
class Person constructor(name: String) {}
class Empty
```

# Classes

- Constructors
  - Primary constructor
    - At class header
    - No code body
    - Parameters

```
class Person constructor(name: String)
// constructor keyword can be omitted
class Person(name: String)
```

# Classes

- Constructors
  - Initializer blocks
    - Execution in order
    - Can use primary constructor parameters

```
class Person(name: String) {
    val property1 = name
    init { print(name) }
    val property1 = name.length
    init { print(name.length) }
}
```

# Classes

- Properties
  - Declaration
  - Default values
  - `val` or `var`

```
class Person(val name: String, var
age: Int = 30)
```

# Classes

- Secondary constructors
  - class body

```
class Person(val name: String) {
    init { print("") }
    constructor(s: String):this(s) {
    }
}

class Person {
    // Implicitly executed
    init { print("") }
    constructor(name: String) {}
}
```

# Classes

- Instances
- Abstract classes
  - Cannot be instantiated

```
// Class instances
val person = Person()
val person = Person("Ali")

// Abstract
abstract class Shape {
    abstract fun draw()
    fun fill() {}
}
class Rectangle : Shape() {
    override fun draw() {}
}
```

# Classes

- Companion objects
  - Members are called using the class name

```
class Person {
    companion object {
        fun newInstance(): Person {
            return Person()
        }
    }
}

val instance = Person.newInstance()
```

# Inheritance

- Every class is-an `Any`
- Classes are final by default
- `open` for inheritance
- Explicit supertype

```
// Implicitly inherits from Any
class Person {}


open class Person(name: String)


class Employee(name: String) :
Person(name)
```

# Inheritance

- Overriding methods
  - A class must be `open`
  - Overridable methods must be `open`
  - Override method must use `override`
- Q: Overloading?

```
open class Shape {
    open fun draw() {}

    // Cannot be overridden
    fun fill() {}
}

class Circle() : Shape() {
    override fun draw() {}
}
```

# Inheritance

- Overriding properties
  - Same as in methods
- `var` overrides `val`
  - not vice versa

```
open class Shape {
    open val origin: Int = 0
}

class Rectangle : Shape() {
    override var origin = 4
}
```

# Inheritance

- Calling from superclass
  - `super` keyword

```
open class Rectangle {
    open fun draw() {}
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
    }
}
```

# Inheritance

- Interfaces
  - Abstract and non-abstract methods
  - Abstract properties
- Q: Abstract class?

```
interface Named {
    val name: String // abstract

    fun bar()
    fun foo() {
        print(name)
    }
}
```

# Inheritance

- Interfaces Inheritance
  - Inheritance from multiple interfaces
  - Interface Inheritance from interface
- Q: Inheritance from multiple `class` type?

```
interface IA {
    fun foo()
    fun bar()
}

interface IB: IA {
    override fun bar() {}
}

class C : IB {
    override fun foo() {}
}
```

# Data classes

- `data` keyword
- Requirements
  - Primary constructor has at least 1 parameter
  - Primary constructor parameters must be marked as `val` or `var`
  - Cannot be `abstract` or `open`
- Automatic derivation
  - `toString()`
  - `copy()`

```
data class Person(val name: String,
val age: Int)

println(person.toString()) //
Person(name=John, age=42)


val person = Person("Ali", 1)

val employee = person.copy(age = 2)
```

# Visibility modifiers

- Access restriction
  - `public`, `private`, `internal`, `protected`
- Packages
  - `public`: everywhere, default
  - `private`: in-file only
  - `protected`: N/A
  - `internal`: in-module only

```
package foo


fun baz() {}

class Bar {}
```

# Visibility modifiers

- Class members
  - `public`: everywhere, default
  - `private`: in-class only
  - `protected`: in-class, in-subclass
  - `internal`: in-module only

```
open class A {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4  // public by default
}
```

```
class B : A() {
    // a is not visible

    override val b = 5
    override val c = 7
}

class C(a: A) {
    // o.a, o.b are not visible
    val c = o.c + o.d
}
```

# Collections

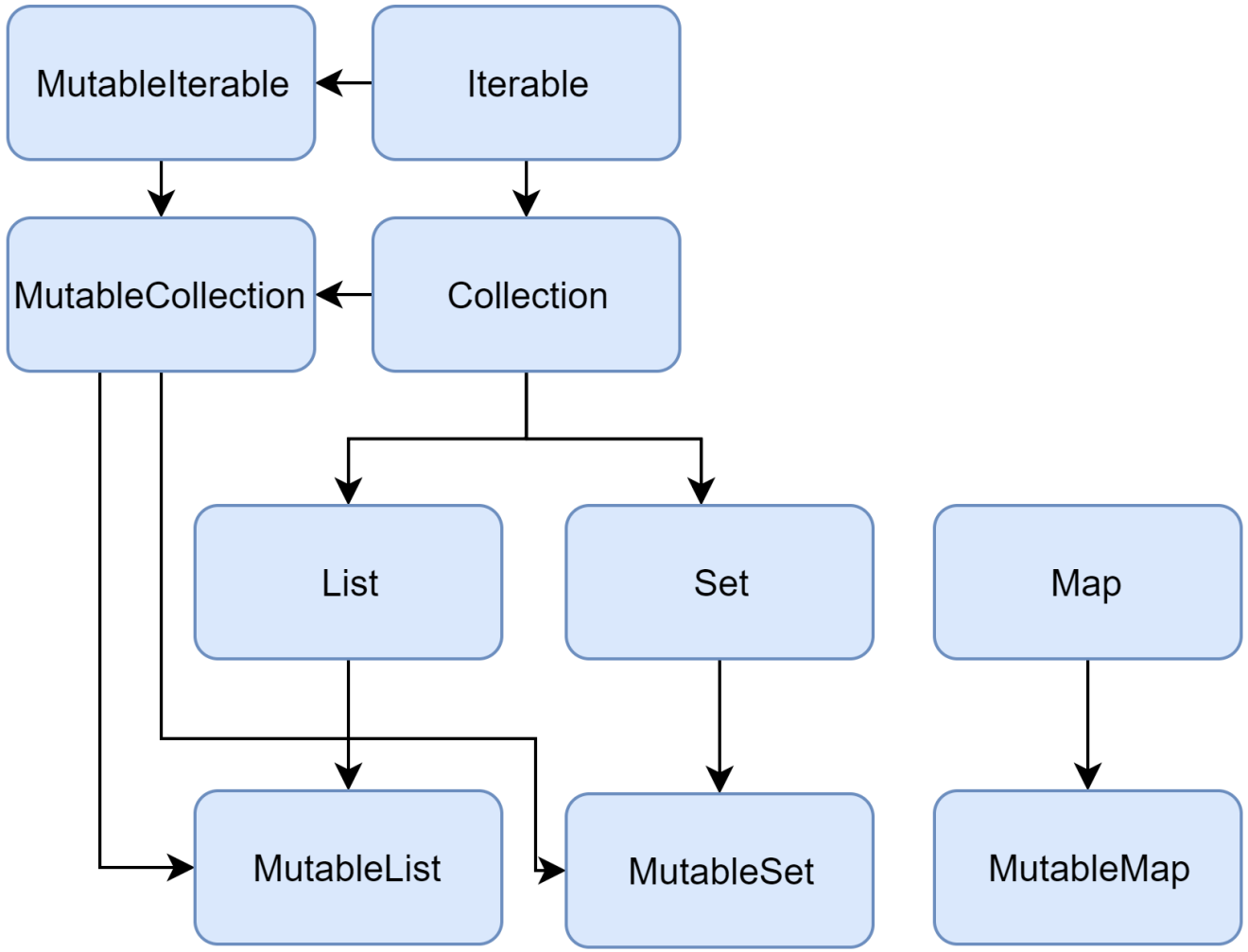
- Groups of 0 or more items
- Types
  - List
  - Set
  - Map
- Read-only or mutable

# Collections

- Read-only
  - Access operations
  - `String` was immutable. Remember?
- Mutable
  - Extends read-only interface
  - Write operations
  - Q: `var`? or `val`?

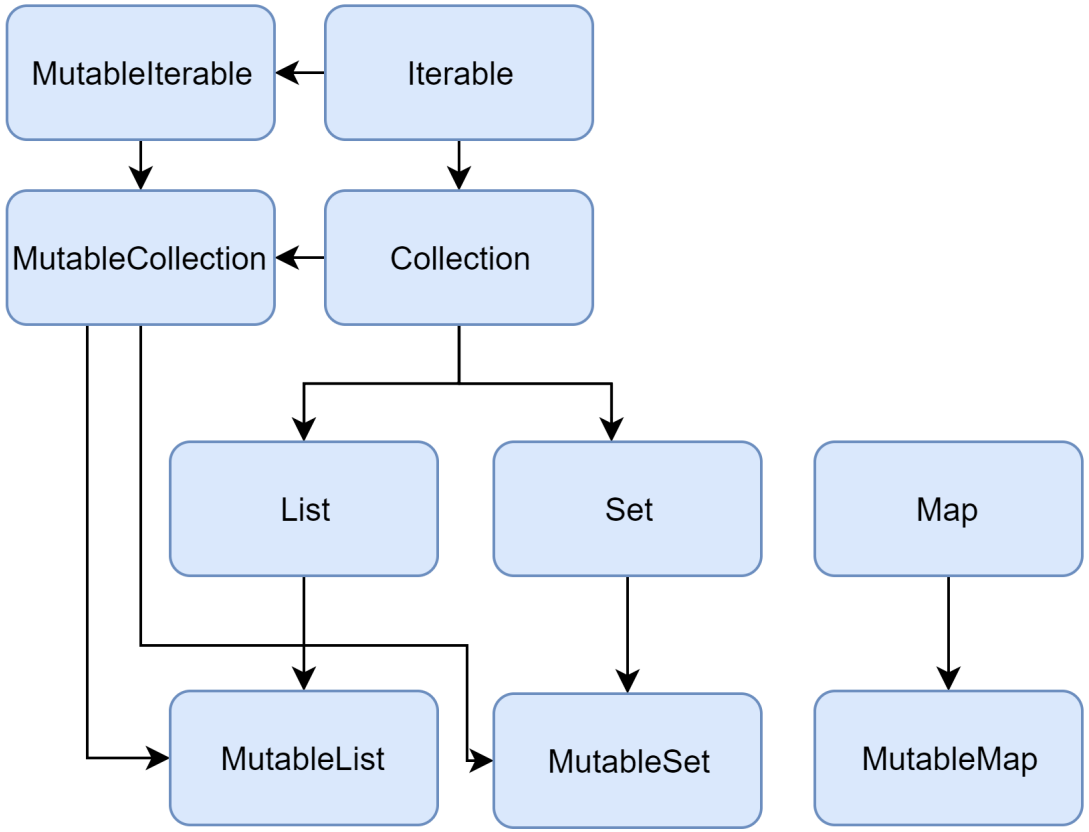
```
// Alter a val list
val list = mutableListOf("o", "t")
list.add("f")
```

# Collections



# Collections

- Collection
  - Root of List & Set
  - Read-only operations
    - `size`
    - `contains(element)`
- MutableCollection
  - Is-a Collection
  - Write operations
    - `add(element)`
    - `remove(element)`



# Collections

- List
  - Ordered
  - Indexed access
  - Elements may repeat
  - Equality: size, elements, positions
- MutableList
  - Is-a List
  - write operations

```
val rect = Rectangle()
val circ = Circle()
val list1 = listOf(rect, circ)
val list2 = listOf(rect, circ)
// list1 and list2 are equal

val list = mutableListOf(1, 2)
list.removeAt(index)
list.shuffle()
list[index] = 5
```

# Collections

- Set
  - Unique elements
  - Undefined order
  - Equality: size, elements, positions
- MutableSet
  - Is-a Set
  - Write operations
  - Preserves order of insertion

```
val set1 = setOf(1, 2, 3, 4)
val set2 = setOf(4, 3, 2, 1)


println(set1.first() == set2.last())
// true
```

# Collections

- Map
  - Set of key-value pairs or entries
  - Keys are unique
  - Key maps to 1 value
  - Values may repeat
    - `in`, `values`, `containsValue()`
  - Equality: entries
- MutableMap
  - Is-a Map
  - Write operations
  - Preserves order of insertion

```
val map1 = mapOf("key1" to 1, "key2" to 2)
val map2 = mapOf("key2" to 2, "key1" to 1)
// map1 and map2 are equal


val map = mutableMapOf("key1" to 1, "key2" to 2)
map.put("key", 3)
map["key"] = 3
```

# Collections Construction

- From elements
  - `listOf<T>()`, `mutableListOf<T>()`
  - `setOf<T>()`, `mutableSetOf<T>()`
  - `mapOf()`,`mutableMapOf()`
- Empty collections
  - `emptyList()`
  - `emptySet()`
  - `emptyMap()`

```
val set = setOf(1, 2, 3, 4)


val empty = mutableSetOf<String>()
val map = mutableMapOf<String, String>()


val map = mutableMapOf<String, String>().apply {
    this["one"] = "1"; this["two"] = "2"
}


val list = emptyList<String>()
```

# Collections Construction

- Builder functions
  - `buildList()`
  - `buildSet()`
  - `buildMap()`

```
// map is read-only
val map = buildMap {
    // this is MutableMap<String, Int>
    put("a", 1)
    this["c"] = 4
}
```

# Collections Construction

- Copying functions
  - `toList()`, `toMutableList()`,`toSet()`
  - Shallow copy
  - Not affected by add/remove
  - Convert between types
- New reference
  - Same collection
  - Affected by add/remove

```
val list = mutableListOf(1, 2, 3)
val set = list.toMutableSet()
set.add(3)
set.add(4) // Guess?

val list1 = mutableListOf(1, 2, 3)
val list2 = list1
list2.add(4)
```

# Conditions

- Expression vs statement
- `if`

```
val a = 2; val b = 3
var max = a

if (a < b) max = b

if (a > b) {
    max = a
} else {
    max = b
}

max = if (a > b) a else b
```

# Conditions

- `when`
  - Multiple branches
  - Sequential matching
  - else may be mandatory:
    - when is an expression
    - non-covered cases
  - Combined conditions

```
when (x) {
    1 -> print("x == 1")
    2, 3 -> print("x is 2 or 3")
    s.toInt() -> print("x == s")
    else -> {
        print("Otherwise")
    }
}
```

# Loops

- `for`
  - iterates through any iterator
    - `next()`
    - `hasNext(): Boolean`
  - iterates using index

```
for (item in collection) {
    print(item)
}

for (i in array.indices) {
    println(array[i])
}
```

# Loops

- `while` and `do-while`
  - execute  while their condition is true.
  - `while` checks the condition first
  - `do-while` executes the body first

```
while (x > 0) {
    x--
}

do {
    val y = retrieveData()
} while (y != null)
```

# Loops

- Jump expressions
  - `return` returns from the nearest function.
  - `break` terminates the nearest loop.
  - `continue` proceeds to the next step of the nearest loop.

```
for (i in 1..100) {
    for (j in 1..100) {
        if (...) break
        if (...) continue
    }
}
```

# References

- Classes https://kotlinlang.org/docs/classes.html
- Inheritance https://kotlinlang.org/docs/inheritance.html
- Data classes https://kotlinlang.org/docs/data-classes.html
- Collections: https://kotlinlang.org/docs/collections-overview.html
- Control flow: https://kotlinlang.org/docs/control-flow.html

- Kotlin Playground: https://play.kotlinlang.org