Kotlin Grammar

Mobile App Programming
Fall, 2024

https://play.kotlinlang.org

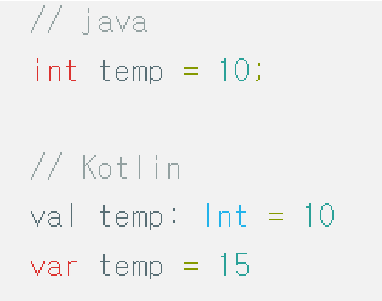
What is Kotlin?

- A cross-platform, statically typed, JVM-targeted programming language.
  - **Cross-platform** : the system or the product can work across multiple platforms or operating system environments.
  - **Statically typed** : the variables used in the program must explicitly be declared along with their types (data type).
- Less code combined with greater readability.
- Interoperability with Java.
- Kotlin support in Android Jetpack and other libraries, such as coroutines.
- Built-in null safety support.

Today, learn about Kotlin Basic Syntax and rule.

Basic syntax

- Unlike programming languages such as Java and C++, Kotlin does not require semi-colons at the end of each statement or expression line.



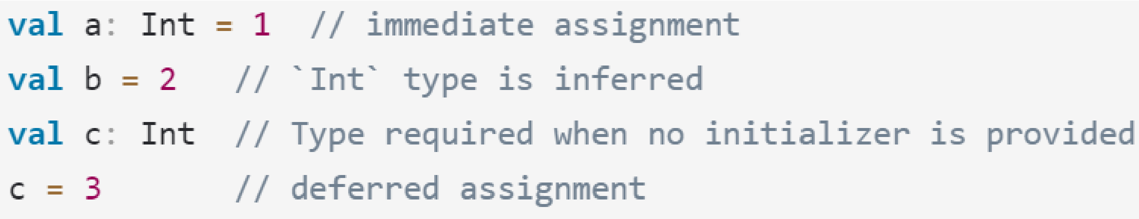
```
// java
int temp = 10;

// Kotlin
val temp: Int = 10
var temp = 15
```

- Semi-colons are only required when multiple statements appear on the same line

```
val mynumber = 10; println(mynumber)
```

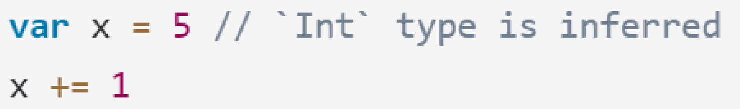
Variable Declaration

- Read-only local variables are defined using the keyword **val**. They can be assigned a value only once.



```
val a: Int = 1  // immediate assignment
val b = 2   // `Int` type is inferred
val c: Int  // Type required when no initializer is provided
c = 3       // deferred assignment
```

- Variables that can be re-assigned use the **var** keyword.



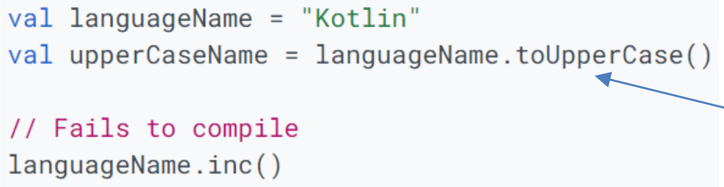
```
var x = 5 // `Int` type is inferred
x += 1
```

- Ref) https://kotlinlang.org/docs/properties.html

Type inference

- Continuing the previous example, when you assign an initial value to **b**, the Kotlin compiler can infer the type based on the type of the assigned value.
- Note that Kotlin is a statically-typed language.
- This means that the type is resolved at compile time and never changes.
- Example)

‘String’ type is inffered



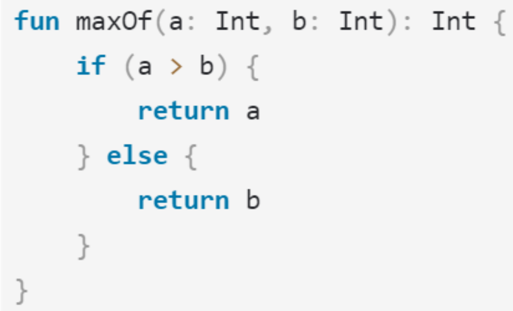
```kotlin
val languageName = "Kotlin"
val upperCaseName = languageName.toUpperCase()

// Fails to compile
languageName.inc()
```

‘String’ method can work

‘Int’ method can not work

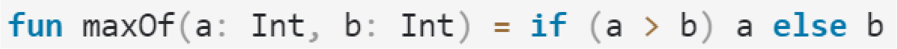
Conditional expressions – If expression

- The most common of these is an if-else statement.



```kotlin
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

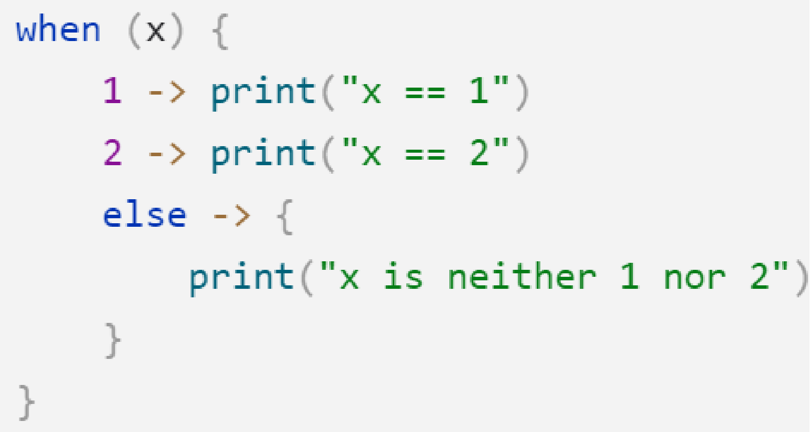
- In Kotlin, **if** can also be used as an expression.



```kotlin
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

Conditional expressions – When expression

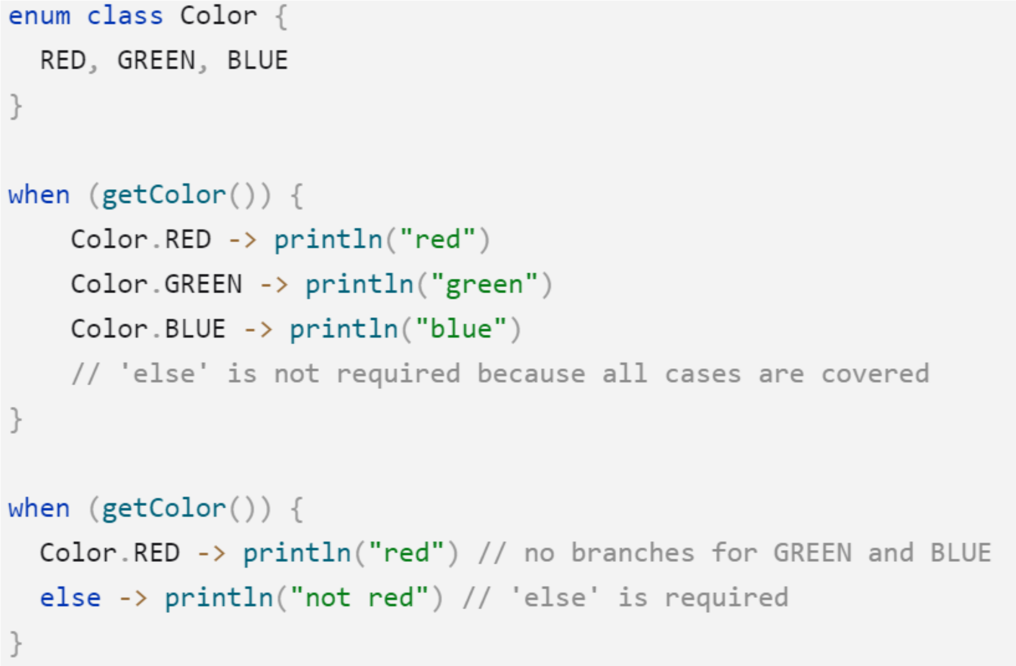
- **when** defines a conditional expression with multiple branches. It is similar to the switch statement in C-like languages.



```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

Conditional expressions – When expression

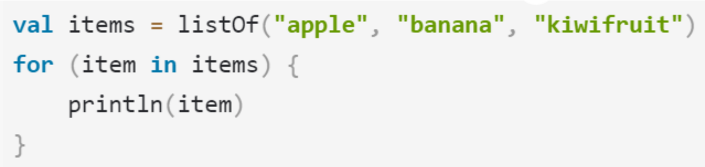
- In **when** statements, the else branch is mandatory in the following conditions:
  - **when** has a subject of an **Boolean**, **enum**, or **sealed** type, or their nullable counterparts.
  - branches of when don't cover all possible cases for this subject.



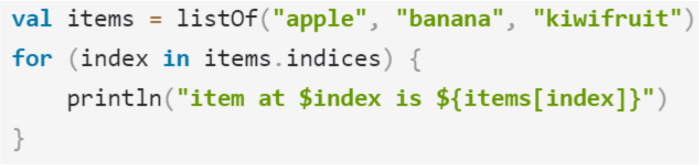
```
enum class Color {
  RED, GREEN, BLUE
}

when (getColor()) {
    Color.RED -> println("red")
    Color.GREEN -> println("green")
    Color.BLUE -> println("blue")
    // 'else' is not required because all cases are covered
}

when (getColor()) {
  Color.RED -> println("red") // no branches for GREEN and BLUE
  else -> println("not red") // 'else' is required
}
```

Loop expressions – For Loops

- The for loop iterates through anything that provides an iterator. This is equivalent to the foreach loop in languages like C#.

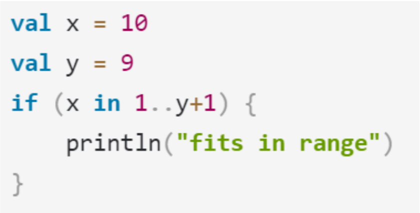


```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

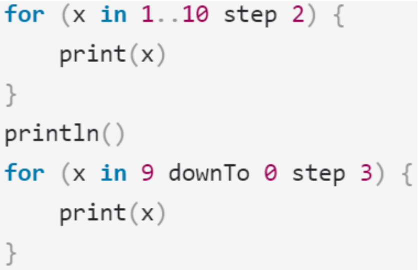


```
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
}
```

- Check if a number is within a range using in operator.



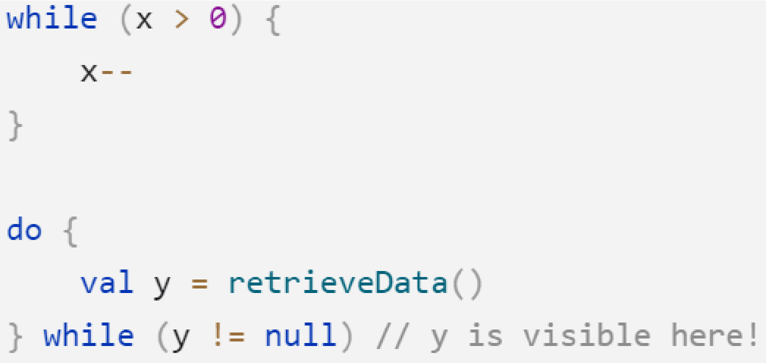
```
val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}
```



```
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

Loop expressions – While Loops

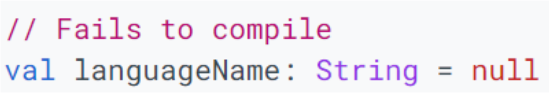
- while and do-while loops execute their body continuously while their condition is satisfied.
  - while checks the condition and, if it's satisfied, executes the body and then returns to the condition check.
  - do-while executes the body and then checks the condition. If it's satisfied, the loop repeats. ranches of when don't cover all possible cases for this subject.



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

do {
    val y = retrieveData()
} while (y != null) // y is visible here!
```

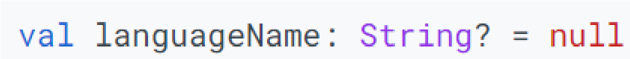
Nullable values and null checks (Null safety)

- Kotlin variables can't hold null values by default.



```
// Fails to compile
val languageName: String = null
```

- A reference must be explicitly marked as nullable when **null** value is possible. Nullable type names have **?** at the end.



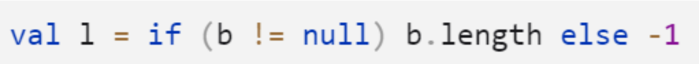
```
val languageName: String? = null
```

- You must handle nullable variables carefully or risk a dreaded NullPointerException.

- In Java, for example, if you attempt to invoke a method on a null value, your program crashes.

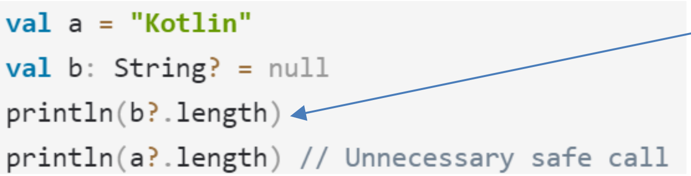
Null Safe call operator - ?.

- There are two ways to access properties on a nullable variable:
  - 1) Checking for null in conditions



```
val l = if (b != null) b.length else -1
```

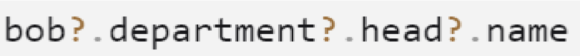
  - 2) Use the safe call operator **?.**



```
val a = "Kotlin"
val b: String? = null
println(b?.length)
println(a?.length) // Unnecessary safe call
```

This returns **b.length** if **b** is not null, and **null** otherwise.

  - Safe calls are useful in chains also.

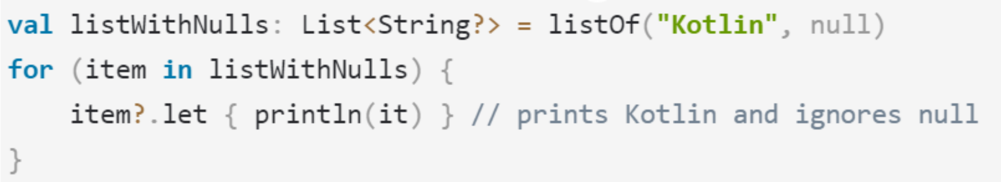


```
bob?.department?.head?.name
```

Such a chain returns **null** if any of the properties in it is **null**.

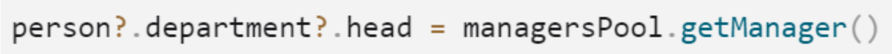
Null Safe call operator - ?.

- To perform a certain operation only for non-null values, you can use the safe call operator together with **let**.



```
val listWithNulls: List<String?> = listOf("Kotlin", null)
for (item in listWithNulls) {
    item?.let { println(it) } // prints Kotlin and ignores null
}
```

- A safe call can also be placed on the left side of an assignment.

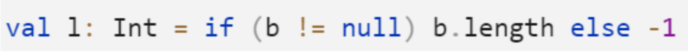


```
person?.department?.head = managersPool.getManager()
```

If either `person` or `person.department` is **null**, the function is not called:

Elvis operator - ?:

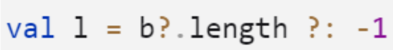
- When you have a nullable reference, b, you can say "if b is not null, use it, otherwise use some non-null value"



```
val l: Int = if (b != null) b.length else -1
```

In 11p, example way 1)

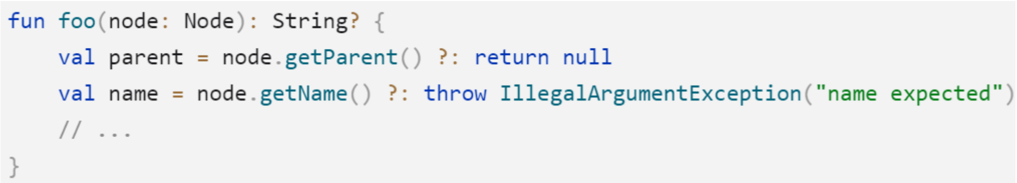
- you can also express this with the Elvis operator **?:**



```
val l = b?.length ?: -1
```

If b?.length is not **null**, the Elvis operator returns it ,otherwise it returns the expression to the right, -1.

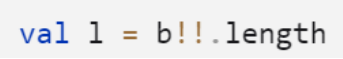
- Since **throw** and **return** are expressions in Kotlin, they can also be used on the right-hand side of the Elvis operator.



```
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

The !! Operator && Safe cast operator as?

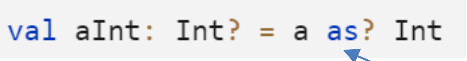
- The not-null assertion operator (**!!**) converts any value to a non-null type and throws an exception if the value is **null**.



```
val l = b!!.length
```

If b!!.length is not **null**, this will returns a non-null value of b
, otherwise it throw an NPE(Null Point Exception).

- To avoid exceptions, use the safe cast operator **as?**, which returns **null** on failure.

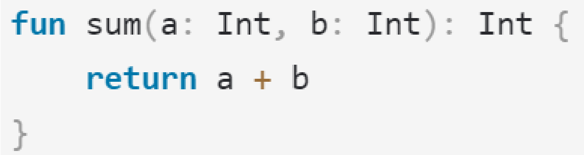


```
val aInt: Int? = a as? Int
```

null cannot be cast to **Int**, as this type is not nullable.
If a is null, the code above throws an exception.

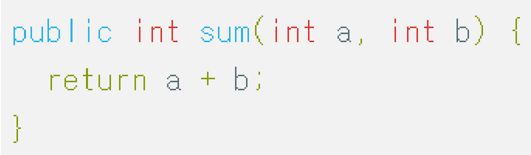
Functions

- A function example with two Int parameters and Int return type.



```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

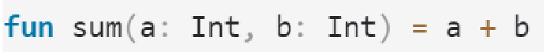
Kotlin example



```
public int sum(int a, int b) {
  return a + b;
}
```

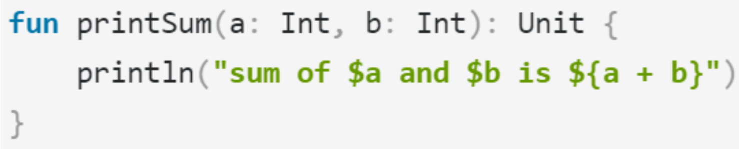
Java example

- A function body can be an expression. Its return type is inferred.



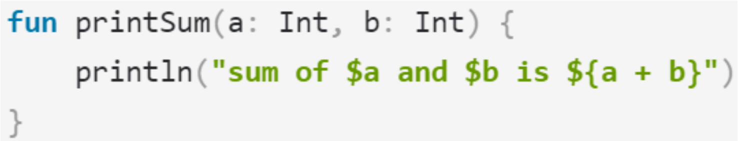
```
fun sum(a: Int, b: Int) = a + b
```

- A function that returns no meaningful value.



```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

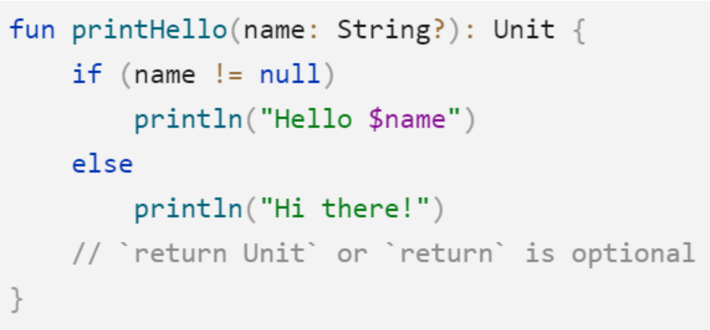
- Unit return type can be omitted.



```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

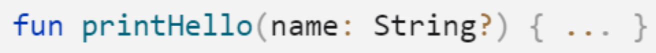
Unit-returning functions & Single-expression functions

- If a function does not return a useful value, its return type is **Unit**. **Unit** is a type with only one value – **Unit**, which does not have to be returned explicitly:



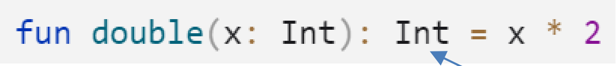
```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

- The Unit return type declaration is also optional. Above code is equivalent to:

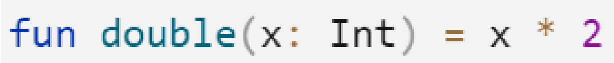


```
fun printHello(name: String?) { ... }
```

- When a function returns a single expression, the curly braces can be omitted and the body is specified after a = symbol:



```
fun double(x: Int): Int = x * 2
```



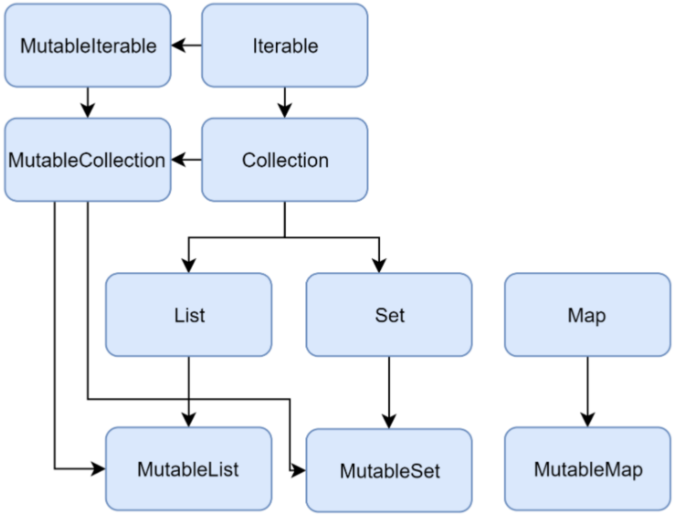
```
fun double(x: Int) = x * 2
```

'Explicitly declaring the return type is optional ,
when this can be inferred by the compiler

Collections overview

- The Kotlin Standard Library provides implementations for basic collection types: sets, lists, and maps
  - read-only interface : provides operations for accessing collection elements
  - mutable interface : extends the corresponding read-only interface with write operations: adding, removing, and updating its elements

- Below is a diagram of the Kotlin collection interfaces:



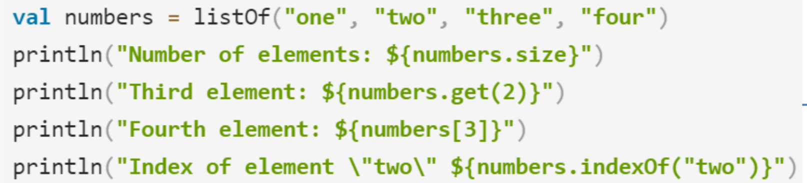
- List: an ordered collection with access to elements by indices

- Set: a collection of unique element

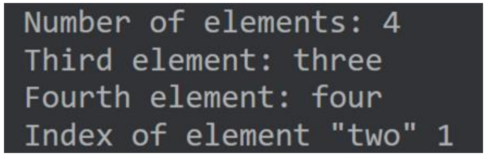
- Map: a set of key-value pairs

List

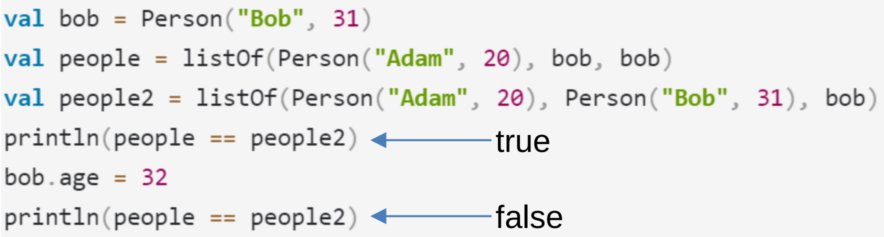
- List<T> stores elements in a specified order and provides indexed access to them. Indices start from 0



```
val numbers = listOf("one", "two", "three", "four")
println("Number of elements: ${numbers.size}")
println("Third element: ${numbers.get(2)}")
println("Fourth element: ${numbers[3]}")
println("Index of element \"two\" ${numbers.indexOf("two")}")
```



```
Number of elements: 4
Third element: three
Fourth element: four
Index of element "two" 1
```

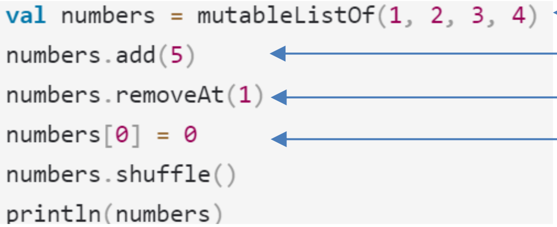


```
val bob = Person("Bob", 31)
val people = listOf(Person("Adam", 20), bob, bob)
val people2 = listOf(Person("Adam", 20), Person("Bob", 31), bob)
println(people == people2)
bob.age = 32
println(people == people2)
```

true

false

- MutableList<T> is a List with list-specific write operations, add, remove...



```
val numbers = mutableListOf(1, 2, 3, 4)
numbers.add(5)
numbers.removeAt(1)
numbers[0] = 0
numbers.shuffle()
println(numbers)
```

[1,2,3,4]

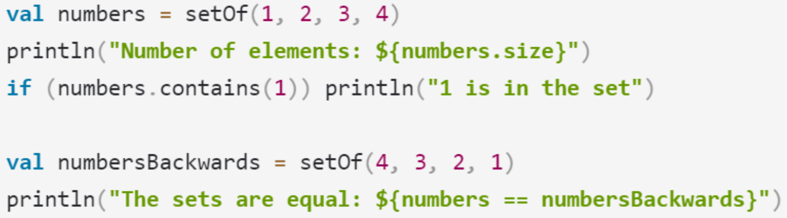
[1,2,3,4,5]

[1,3,4,5]

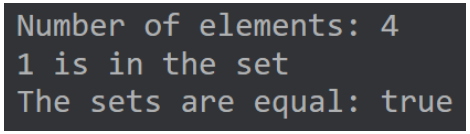
[0,3,4,5]

Set

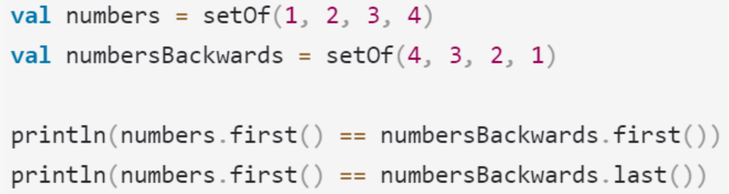
- Set<T> stores unique elements; their order is generally undefined. a Set can contain only one null.



```
val numbers = setOf(1, 2, 3, 4)
println("Number of elements: ${numbers.size}")
if (numbers.contains(1)) println("1 is in the set")

val numbersBackwards = setOf(4, 3, 2, 1)
println("The sets are equal: ${numbers == numbersBackwards}")
```



```
Number of elements: 4
1 is in the set
The sets are equal: true
```



```
val numbers = setOf(1, 2, 3, 4)
val numbersBackwards = setOf(4, 3, 2, 1)

println(numbers.first() == numbersBackwards.first())
println(numbers.first() == numbersBackwards.last())
```

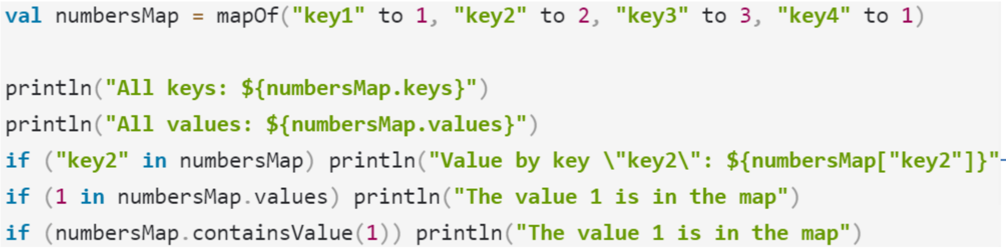
false

true

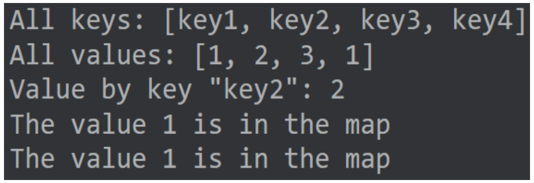
- Mutableset<T> is a Set with write operations from MutableCollection.

Map

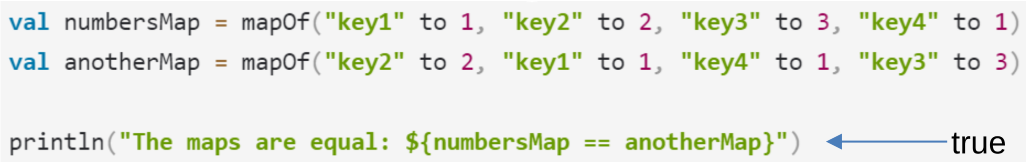
- Map <K,V> is not an inheritor of the Collection interface; however, it's a Kotlin collection type as well.
- A Map stores key-value pairs (or entries); keys are unique, but different keys can be paired with equal values.



```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)

println("All keys: ${numbersMap.keys}")
println("All values: ${numbersMap.values}")
if ("key2" in numbersMap) println("Value by key \"key2\": ${numbersMap["key2"]}")
if (1 in numbersMap.values) println("The value 1 is in the map")
if (numbersMap.containsValue(1)) println("The value 1 is in the map")
```

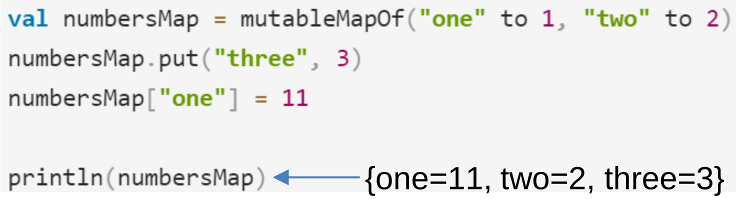


```
All keys: [key1, key2, key3, key4]
All values: [1, 2, 3, 1]
Value by key "key2": 2
The value 1 is in the map
The value 1 is in the map
```



```
val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1, "key3" to 3)

println("The maps are equal: ${numbersMap == anotherMap}")
```

true

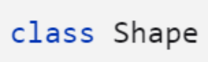


```
val numbersMap = mutableMapOf("one" to 1, "two" to 2)
numbersMap.put("three", 3)
numbersMap["one"] = 11

println(numbersMap)
```

{one=11, two=2, three=3}

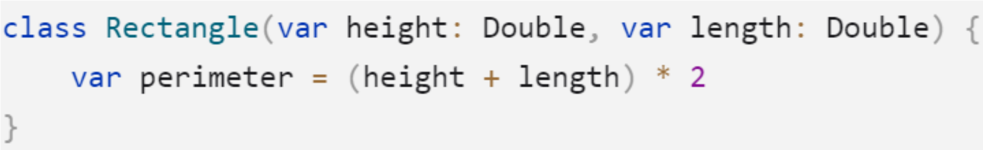
Creating classes and instances

- To define a class, use the class keyword.



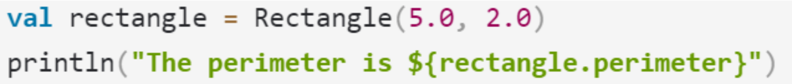
```
class Shape
```

- Properties of a class can be listed in its declaration or body.



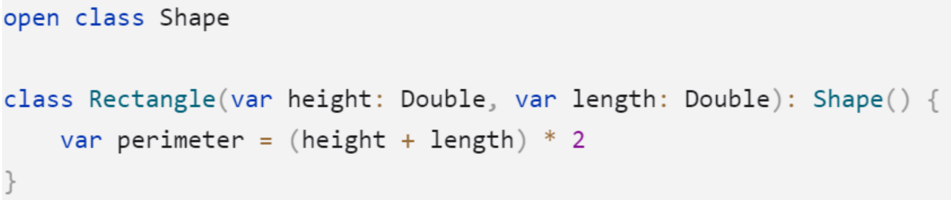
```
class Rectangle(var height: Double, var length: Double) {
    var perimeter = (height + length) * 2
}
```

- The default constructor with parameters listed in the class declaration is available automatically.



```
val rectangle = Rectangle(5.0, 2.0)
println("The perimeter is ${rectangle.perimeter}")
```

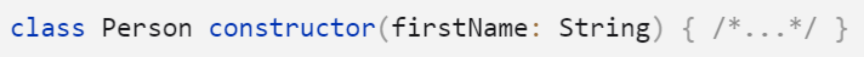
- Inheritance between classes is declared by a colon (:). Classes are final by default; to make a class inheritable, mark it as open.



```
open class Shape

class Rectangle(var height: Double, var length: Double): Shape() {
    var perimeter = (height + length) * 2
}
```

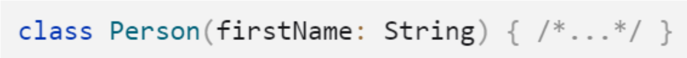
Classes- Constructors

- If the Primary Constructor does not have any annotations or visibility modifiers, the constructor can be omitted



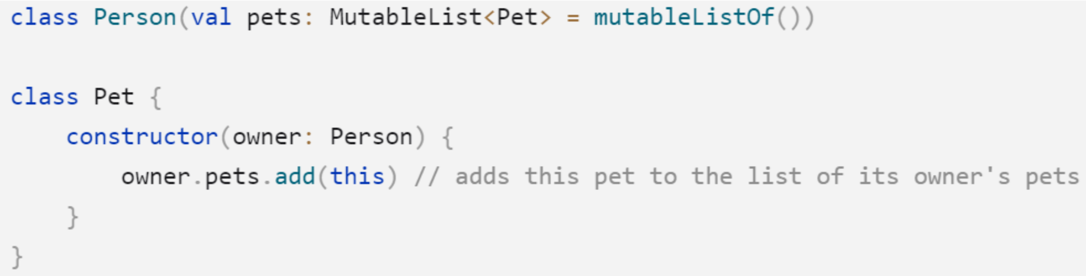
```
class Person constructor(firstName: String) { /*...*/ }
```

↔



```
class Person(firstName: String) { /*...*/ }
```

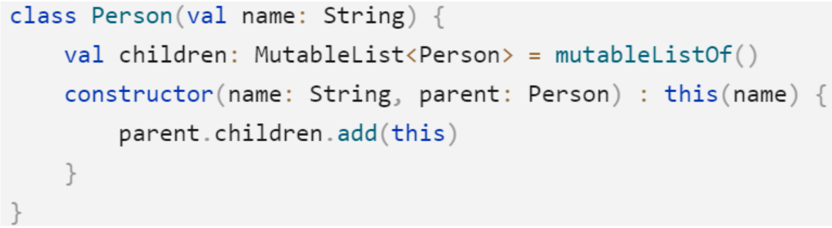
- A class can also declare secondary constructors, which are prefixed with constructor.



```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

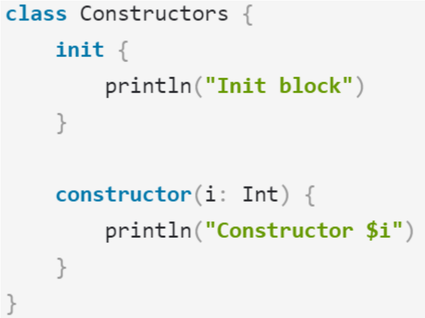
Classes- Constructors

- If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor
  - Delegation to another constructor of the same class is done using the this keyword:



```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

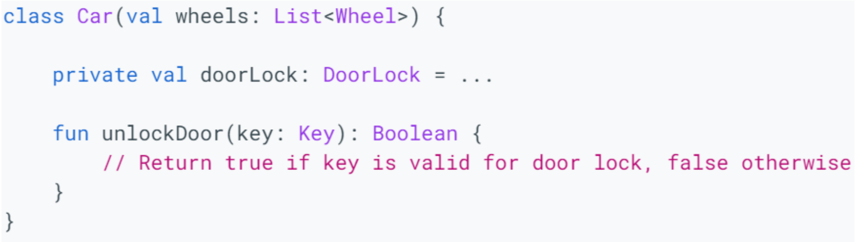
- Even if the class has no primary constructor, the delegation still happens implicitly, and the initializer blocks are still executed:



```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
```

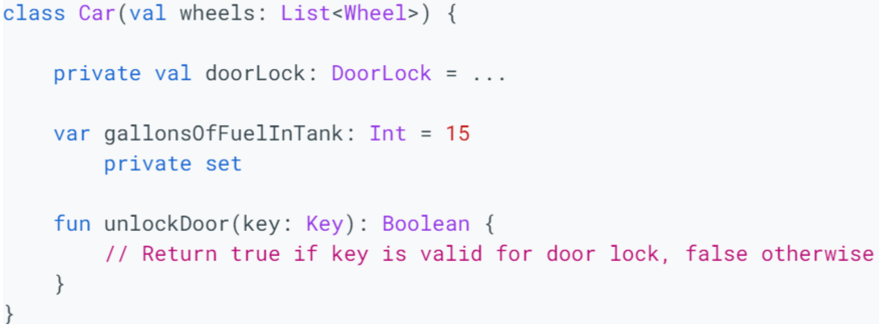
Class functions and encapsulation

- Classes use functions to model behavior. Functions can modify state, helping you to expose only the data that you wish to expose. This access control is part of a larger object-oriented concept known as encapsulation.



```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
}
```

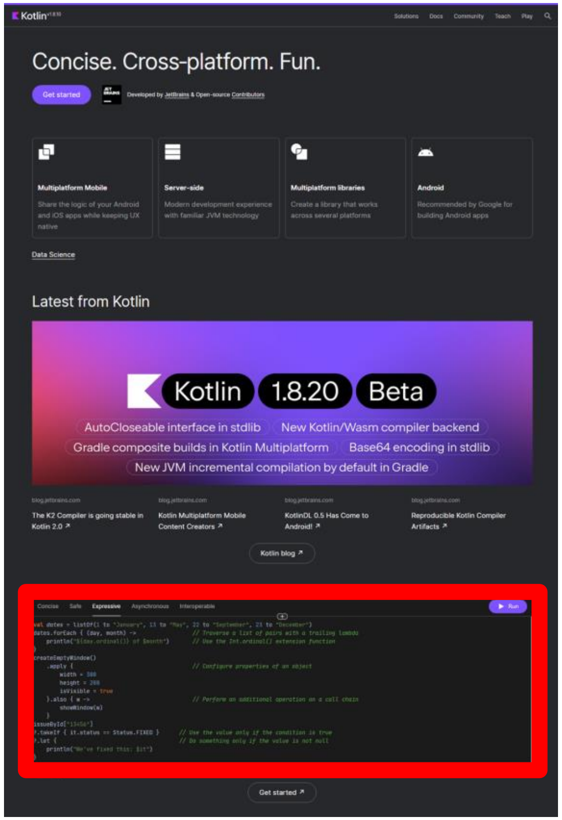
- If you would like to customize how a property is referenced, you can provide a custom getter and setter.



```
class Car(val wheels: List<Wheel>) {

    private val doorLock: DoorLock = ...

    var gallonsOfFuelInTank: Int = 15
        private set

    fun unlockDoor(key: Key): Boolean {
        // Return true if key is valid for door lock, false otherwise
    }
}
```

[Lab-Practice #2] Simple Prime number count

- Print all prime numbers under 100.
- You can use open editor in https://kotlinlang.org/



Concise. Cross-platform. Fun.

Developed by JetBrains & Open-source Contributors

Multiplatform Mobile
Share the logic of your Android and iOS apps while keeping UX native

Server-side
Modern development experience with familiar JVM technology

Multiplatform libraries
Create a library that works across several platforms

Android
Recommended by Google for building Android apps

Data Science

Latest from Kotlin

Kotlin 1.8.20 Beta

AutoCloseable interface in stdlib
New Kotlin/Wasm compiler backend
Gradle composite builds in Kotlin Multiplatform
Base64 encoding in stdlib
New JVM incremental compilation by default in Gradle

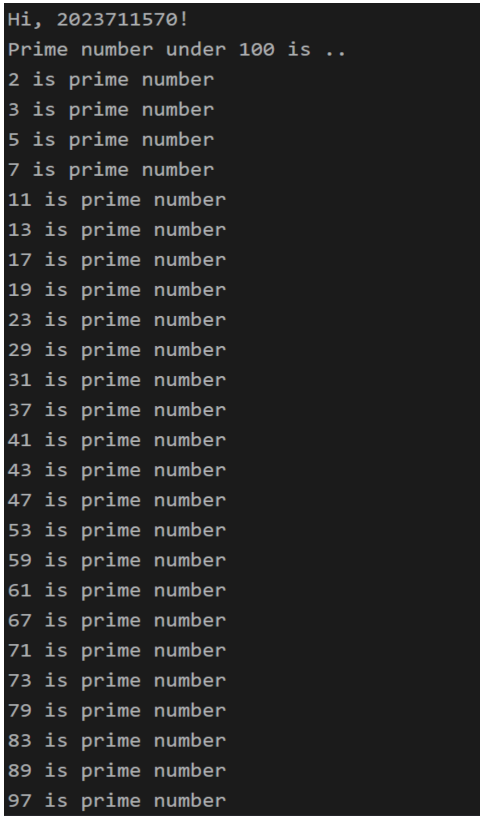
blog.jetbrains.com
The K2 Compiler is going stable in Kotlin 2.0 ↗

blog.jetbrains.com
Kotlin Multiplatform Mobile Content Creators ↗

blog.jetbrains.com
KotlinDL 0.5 Has Come to Android! ↗

blog.jetbrains.com
Reproducible Kotlin Compiler Artifacts ↗

```
val dates = listOf(1 to "January", 13 to "May", 22 to "September", 23 to "December")
dates.forEach { (day, month) ->                  // Traverse a list of pairs with a trailing lambda
    println("${day.ordinal()} of $month")        // Use the Int.ordinal() extension function
}
createEmptyWindow()
    .apply {                                     // Configure properties of an object
        width = 300
        height = 200
        isVisible = true
    }.also { w ->                                // Perform an additional operation on a call chain
        showWindow(w)
    }
issueById["13456"]
?.takeIf { it.status == Status.FIXED }      // Use the value only if the condition is true
?.let {                                     // Do something only if the value is not null
    println("We've fixed this: $it")
}
```



```
Hi, 2023711570!
Prime number under 100 is ..
2 is prime number
3 is prime number
5 is prime number
7 is prime number
11 is prime number
13 is prime number
17 is prime number
19 is prime number
23 is prime number
29 is prime number
31 is prime number
37 is prime number
41 is prime number
43 is prime number
47 is prime number
53 is prime number
59 is prime number
61 is prime number
67 is prime number
71 is prime number
73 is prime number
79 is prime number
83 is prime number
89 is prime number
97 is prime number
```

[Lab-Practice #2] Simple Prime number count

- You must print your student number.
- You must use Kotlin language.
- Before you leave the class, please check your example application.