Operações com char

Char+/-Int = Char (Int+Char -> erro)
Char-Char=Int / Char+Char->erro

Data class – tipo agregado

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

Enum Class – tipo com certo nºde objetos possíveis

```
enum class Color(val rgb: Int) {
   RED(0xFF0000), GREEN(0x00FF00)}
```

Dir. RED – selecionar objeto RED
RED.name – nome do objeto RED
RED.ordinal – index do objeto RED
Color.values() – array de objetos de Color

Interface - declarations of abstract methods, as well as method implementations

```
interface MyInterface {
   fun bar()
   fun foo() {// optional body}
}
```

Abstract class - does not have implementations

```
abstract class Polygon {
   abstract fun draw()
}
```

Companion Object - an object declaration inside a class

```
class MyClass {
   companion object Factory {
      fun create(): MyClass = MyClass()
   }
}
val instance = MyClass.create()
val x = MyClass.Companion
```

Keywords

step / in / !in / until / downTo
class / data class / enum class / abstract class / interface
companion object / object
init / constructor / sealed / open
override / private / suspend / fun
this / it
const / val / var / varagr / lateinit
if / else if / else / when
for / while / do while / repeat
is / !is / return / as
break / continue / throw
typealias / typeof / by / get() / set()

Null

:Type? -> nullable
ball=ball?.função() -> se ball for null, retorna null; senão for, executa a função
val pos = ball?.pos ?: 10 -> Se ball for null, retorna 10; senão for, retorna pos
!! tem-se a certeza que não é null

Random

```
b.apply{code block}
b.also{}
b.let{}
typealias View = (Session) -> Unit
var counter: Int = 0
   get() = field
   set(newValue:Int){field=newValue}
```

Iterables

```
val nums:List<Int> =listOf(1,3,2,4)
nums.first() // 1
nums.last() //4
nums.singleOrNull() // null
nums.indices //0..3
nums.map{num-> code block}
nums.filter{ condition }
nums. firstOrNull{it == 1} /1
nums.forEach{instrução}
nums.any{ condition }
nums.all{ condition }
nums.none{ condition }
nums.takeLast(2) // [2, 4]
nums.take(2) // [1, 3]
```

Compose for Desktop

@Composable functions só podem ser chamadas por outras @Composable

```
val a=remember
{mutableStateOf<Type>(initialValue)}

Image(painter=
painterResource("imagePath"),
contentDescription = null)

Text(text="",fontSide=2.sp)
Button(onClick = {}) {content}
```

Canvas - you can draw elements
Column – place items vertically
Row - place items horizontally
Box - put elements on top of another

```
application {} – especial(não é
chamado por outro composable)
Window(onCloseRequest:()->Unit,
title) – displays a window
MaterialTheme {content}
LaunchedEffect(key1){content} – é
recomposto quando key1 muda
AlertDialog(onDismissRequest,button
s)

val textState=remember
{mutableStateOf(TextFieldValue())}
TextField(
value = textState.value,
onValueChange={textState.value=it})

MenuBar{Menu("File"){
  Item(texto, omClick, enabled)
  CheckboxItem(texto, checked,
onCheckedChange)}}
```

Corrotinas

```
val cs = rememberCoroutineScope()
```

cs.launch{code block} - lança uma nova corrotina sem bloquear a thread, enquanto está a ser composto
fun suspend - só pode ser chamada dentro de uma corrotina
delay(millis:Long) – interrompe a corrotina

Tests

@Test – identifica um teste

```
assertTrue(actual)
assertFalse(actual)
assertEquals(expected, actual)
assertContains(array/map, value/key)
assertNull(actual)
assertNotNull(actual)
assertContentEquals(expected,
actual)
assertFailsWith<Exception> {code
block}
```

Exceptions

```
try {} catch (e: ExceptionType) finally{}

class MyException(override val
message: String): Exception(message)

throw Throwable/Exception
```

require(condition){message} - Illegal Argument (requireNotNull)
check(condition) {message} – Illegal State
assert(condition) {message} – Assertation Error

Princípios e Técnicas

Encapsulamento – restrição de acesso direto a algumas propriedades do objecto; é usado para esconder valores/estados da estrutura de dados, para prevenir acesso direto
KISS – keep it simple, stupid; valorizar simplicidade
Testabilidade
Mutabilidade/Imutabilidade
Non Primitive Type Obcession
DRY – do not repeat yourself; reduzir repetição de Código
SRP- single responsability principle; cada responsabilidade deve ser atribuída a uma e sou uma entidade
Open-Close Principle – entidades devem ser abertas para extensão, mas fechadas para modificação
Model View Controler – model é responsável pelo behavior/dados /domain; view é o que o utilizador vê; controller é o que interage entre o model e o view.