# Linguagem Kotlin Desenvolvimento Android

Apresentação: Prof. Carlos Alberto

# TIPOS DE DADOS

- Em Kotlin, tudo é um objeto, diferentemente de Java, em que existia uma diferenciação de tipo de dados primitivos e objetos. Na prática, isso quer dizer que todas as variáveis e tipos que usamos em Kotlin possuem propriedades e métodos, isto é, todo objeto possui características e comportamentos específicos. Podemos fazer um comparativo com o tipo int do Java e o tipo Int do Kotlin. Em essência, ambos servem para a mesma coisa: guardar números inteiros, porém o funcionamento é diferente. O tipo int do Java é um tipo primitivo, isso quer dizer que é uma variável que somente guarda um valor e não tem nenhum comportamento atrelado a ela. Já o tipo Int do Kotlin é um objeto e, sendo um objeto, além de guardar um valor inteiro ele possui métodos que podem nos auxiliar no desenvolvimento.

# TIPOS DE DADOS

- O Int do Kotlin se assemelha ao tipo Integer também do Java, que é um objeto que contém um wrapper para o tipo primitivo int , então, por dentro, tanto o Int do Kotlin quanto o Integer do Java contêm um tipo primitivo int dentro deles. Um exemplo são os métodos de conversão de tipos. Imagine que você tenha uma variável do tipo Int e precise convertê-la para Double , ou Float , ou até mesmo para String . Para isso, basta utilizar os métodos toDouble() , toFloat() e toString() .

# TIPOS DE DADOS

- Exemplo:
- val x: Int = 10;
- var y: Double = x.toDouble() //Retorna um objeto Double a partir do valor de x
- var z: Float = x.toFloat() //Retorna um objeto Float a partir d o valor de x
  var a: String = x.toString() //Retorna um objeto String a parti r do valor de x

# Dados numéricos

- Os tipos numéricos suportados em Kotlin são muito parecidos com os suportados pelo Java. Existem os tipos Double , Float , Long , Int , Short e Byte , todos eles muito parecidos com os tipos suportados em Java, mas com a principal diferença de que todos são objetos. A seguinte tabela mostra o tipo e a quantidade de bits que cada tipo ocupa na memória

| Tipo | Bit |
|---|---|
| Double | 64 |
| Float | 32 |
| Long | 64 |
| Int | 32 |
| Short | 16 |
| Byte | 8 |

# Dados numéricos

- Uma coisa interessante da linguagem é a utilização do underline para facilitar a legibilidade do código na definição de variáveis numéricas. Por exemplo, se eu definir uma variável com valor de 1 milhão, ela ficaria assim:
- val umMilhao = 1000000
- O que não tem nada de errado, mas fica um pouco difícil de ler qual número é aquele sem contar a quantidade de zeros. Para facilitar a leitura desse código, poderíamos utilizar o caractere _ para fazer uma separação de milhar: val umMilhao = 1_000_000

# Strings e Caracteres

- Uma String é um tipo de dado utilizado para guardar conteúdos do tipo texto. Esse conteúdo deve ser definido utilizando aspas duplas, assim:
  - val texto = "Boa tarde, seja bem-vindo ao sistema"
- Uma das coisas bem legais que a linguagem possui são os templates. Templates de Strings são trechos de código inseridos diretamente em uma String e que têm seu resultado concatenado junto à String. Uma vantagem dos templates é que se evita concatenação de Strings, e a operação de concatenação é custosa ao computador, pois ele deve realocar o texto em novo espaço de memória. E se essa operação for repetida diversas vezes durante o programa, com toda a certeza o desempenho geral do App será comprometido.

# Strings e Caracteres

- Com esse recurso, é possível utilizar o valor de uma variável dentro da String através do **caractere $** . Veja um exemplo:
  - fun main(args: Array<String>) {
  - val nomeUsuario = “Jsilva”
  - val saudacao = “Bem-vindo, $nomeUsuario”
  - println(saudacao) }
  - Execute esse código no play Kotlin e veja o resultado.

# Strings e Caracteres

```
fun main(args: Array<String>) {
val nomeUsuario = "JSilva"
 val saudacao = "Bem-vindo, $nomeUsuario"
println(saudacao) }
```

```
Bem-vindo, JSilva
```

# Strings e Caracteres

- Você pode definir também um texto com várias linhas, basta utilizar o delimitador de 3 aspas duplas """ :
- Conforme exemplo ao lado.

```
fun main(args: Array<String>) {
val nomeUsuario = "JSilva"
 val saudacao = "Bem-vindo, $nomeUsuario"
println(saudacao)

//exemplo de texto com mais de uma linha
val text =  """
              Exemplo   de  texto
              com   mais    de  uma
              linha
            """
    println(text)
}
```

```
Bem-vindo, JSilva

                        Exemplo      de      texto
                        com   mais   de      uma
                        linha
```

# Operadores Booleanos

- O tipo booleano é definido como Boolean , e só pode receber valores true (verdadeiro) ou false (falso). Com tipos booleanos, podemos executar as seguintes operações:

| Operação | Operador |
|---|---|
| E | && |
| OU | || |
| NÃO | ! |

- Uma dado interessante é que o tipo booleano possui funções para essas operações. São elas and , or e not , veja o exemplo a seguir:

# Operadores Booleanos

- **Exemplo:**

```
fun main(args:  Array<String>)  {
                val b1  =   true
                val b2  =   false

                val c1  =   b1.and(b2)  //Retorno   será    false
                val c2  =   b1.or(b2)   //Retorno   será    true
                val c3  =   b1.not()    //Retorno   será    false
                println("$c1    $c2 $c3")
}
```

```
false   true    false
```

# Listas e Arrays

- Quando precisamos armazenar mais de um valor em uma variável, podemos utilizar um **Array** . Um Array é como se agrupássemos várias variáveis em uma única variável. Imagine o cenário em que você precise armazenar a nota de 10 alunos. Nada impede de você criar 10 variáveis e armazenar cada nota em uma variável, mas e se fossem 100 alunos, e se fossem 1000 alunos, seria viável criar mil variáveis? Nesse caso, o uso de um Array é necessário. Você pode definir um **Array** de mil posições e em cada posição armazenar uma nota.

# Listas e Arrays

- Um **Array** sempre terá um tamanho fixo e eu consigo definir e resgatar valores através de seu índice utilizando colchetes ( [ ] ). Veja um exemplo, vou definir um **Array** de inteiros de 4 posições:
  - val arrayInt :Array = arrayOf(1, 2, 3, 4)
- E para acessar ou definir um valor do Array , basta utilizar os colchetes passando sua posição:
- val arrayInt :Array = arrayOf(1, 2, 3, 4)
- //passando o índice 2 para acessar o valor da posição 2 do ar ray val x = arrayInt[2] println(x)

# Listas e Arrays

- Nesse exemplo, foi definido um **Array** de inteiros com 4 valores ( 1,2,3,4 ). Em seguida, foi criada uma variável x que recebe o valor que está na posição 2 deste **Array** . Alguns podem pensar que o valor da segunda posição é justamente o número 2 , mas todo **Array** tem seu início na posição 0, então o valor da posição 2 é 3 , nesse caso.
- No entanto, nem sempre sabemos antecipadamente a quantidade de valores que precisamos armazenar. E se eu precisasse adicionar mais um valor nesse **Array** ? Sendo um **Array** , isto não é possível, e para esses casos usamos listas.

# Listas e Arrays

- As estruturas de listas são parecidas com as estruturas de **Array** , com algumas diferenças. Listas essencialmente são estruturas cujo tamanho não preciso saber antecipadamente e posso adicionar novos valores conforme a necessidade. No entanto, em Kotlin há uma diferenciação para listas mutáveis e listas não mutáveis, isso quer dizer que existe um tipo especifico de listas que aceita a adição de novos valores, as mutáveis, e um tipo de lista que não aceita novos valores, as imutáveis. Para definir uma lista mutável, podemos usar a seguinte sintaxe:
- val lista = mutableListOf(1,2,3,4)

# Listas e Arrays

- **Adicionando valores a uma lista**
- Através da função ADD é possível adicionar valores a uma lista veja o exemplo a seguir:
  - val lista = mutableListOf(1,2,3,4)
  - lista.add(5) //adicionando um novo valor a lista
- Além dessa característica, listas possuem algumas funções muito úteis no dia a dia. Veremos 3 delas, a função first , last e filter .

# Listas e Arrays

- **A função first retorna sempre o primeiro elemento da lista:**
  - val lista = mutableListOf(1,2,3,4)
  - val item = lista.first() //a variável item ficará com valor 1
- A função last retorna o último item da lista:
  - val lista = mutableListOf(1,2,3,4)
  - val item = lista.last() //a variável item ficará com valor 4

# Listas e Arrays

- A função filter aplica um filtro específico na lista, ela é bem bacana e economiza bastantes linhas de código. Vamos supor que nesta mesma lista nós quiséssemos aplicar um filtro e obter somente números pares. Isso é possível com o seguinte código:
  - val lista = mutableListOf(1,2,3,4)
  - val numerosPares = lista.filter { it % 2 == 0 }

# Listas e Arrays

- Este código cria uma nova lista chamada numerosPares somente com os números pares! A função filter realiza uma interação na lista e aplica o filtro de acordo com o código que escrevemos dentro das chaves. Nesse caso, o código verificava se cada elemento da lista dividido por 2 é igual a 0.

# Listas e Arrays

- Para definir uma lista imutável, podemos usar a seguinte sintaxe:
  - val lista = listOf(1,2,3,4)
- Todos os métodos utilizados na lista mutável funcionam em listas imutáveis, com exceção do método add , pois em uma lista imutável não é possível adicionar novos valores.

# ESTRUTURAS DE DECISÃO

- Utilizamos uma estrutura de decisão em nosso programa quando queremos que determinado pedaço de código seja executado somente quando uma condição for satisfeita. Imagine um código de login, que deve receber um usuário e uma senha e verificar se esses dados estão corretos. Somente com usuário e senha corretos o sistema deve ser liberado. Para isso, utilizamos uma estrutura de decisão, nesse caso, uma expressão **if** cairia bem. A expressão **if** é uma estrutura de decisão, ou seja, conseguimos mudar o fluxo do programa de acordo com o resultado de um if . Vamos voltar ao exemplo do login e supor que se a senha for igual a 123 o programa exibe uma mensagem de "Acesso concedido", caso contrário, o programa exibe uma mensagem de "Senha incorreta". O código seria assim:

# ESTRUTURAS DE DECISÃO

```
fun main(args:  Array<String>)  {
                val senha    =   "123"
                if( senha    ==  "123"){
                                 println("Acesso concedido")
                }else{
                                 println("Senha  incorreta")
                }
}
```

```
Acesso  concedido
```

# ESTRUTURAS DE DECISÃO

- Um if não precisa ter necessariamente um else , ele pode ter somente a condição if . Veja o seguinte exemplo em que a estrutura verifica se um valor é maior que outro:

```
fun main(args:  Array<String>)  {
    val a   =   10
    val b   =   5
        if( a   >   b)  {
                println("$a é   maior   que $b")

        }
}
```

```
 10      é       maior   que     5
```

# ESTRUTURAS DE DECISÃO

- O código executa o print caso a variável a seja maior que a variável b , caso contrário, não faz nada. Podemos ainda utilizar um else para indicar o que fazer caso a condição seja falsa, veja:

```
fun main(args:  Array<String>)  {
    val a   =   4
    val b   =   5
        if( a   >   b)  {
                println("$a é   maior   que $b")
        }else
            {   println("$a é   menor   que $b")

}}
```

```
4       é       menor   que    5
```

# ESTRUTURAS DE DECISÃO

- Outra forma interessante de se fazer um if em Kotlin é criar toda a estrutura em uma única linha. Em alguns casos isso pode fazer sentido e economizar algumas linhas de código. Vamos supor que no exemplo anterior eu queira saber qual variável é maior, a ou b . Nesse caso, eu poderia utilizar a mesma estrutura anterior e simplesmente armazenar o resultado em outra variável, mas eu também poderia resumir o código da seguinte maneira:

# ESTRUTURAS DE DECISÃO

```
fun main(args:  Array<String>)  {
    val a   =   4
    val b   =   5
        val maior   =   if( a   >   b)  a   else    b
          println("O maior valor é:"+ maior)

}
```

```
O maior valor é:5
```

# ESTRUTURAS DE DECISÃO

- Nesse caso, o if foi usado como uma expressão e não como um simples controle de fluxo, mas veja como o código ficou claro, ele pode ser lido da seguinte forma: "Se a for maior que b , então maior recebe a , caso contrário, maior recebe b "

# ESTRUTURAS DE DECISÃO

- Outra estrutura  do Kotlin é a estrutura **when** . Usamos o **when** quando precisamos fazer várias verificações em seguida. É claro que isso poderia ser feito com vários if s um atrás do outro, mas a estrutura **when** é própria para esses casos. Em um comparativo com Java, o **when** seria o substituto do **switch** , veja:
- Continua no slide 32

# ESTRUTURAS DE DECISÃO

- **Expressão When-Exemplo-01**

```
fun main() {
val x = 3
when    (x) {
                1   ->  print("x    ==  1")
                2   ->  print("x    ==  2")
                else    ->  {
                                print("x   possui  outro   valor")
                }
}
}
```

# ESTRUTURAS DE DECISÃO

- **Expressão When-Exemplo-02**

```
fun main() {
    cases("Teste")
    cases(1)
    cases(0L)
    cases(MyClass())
    cases("")
}

fun cases(obj: Any) {
    when (obj) {                                    // 1
        1 -> println("One")                         // 2
        "Ola" -> println("Saudações")               // 3
        is Long -> println("Grande")                  // 4
        !is String -> println("Não é uma String")       // 5
        else -> println("Desconhecido")                  // 6
    }
}

class MyClass
```

# ESTRUTURAS DE DECISÃO

- **Expressão When-Exemplo-03**
- Se eu quiséssemos, por exemplo, testar se a variável **x** tem valor 1 ou 2, poderia fazer assim:

```
fun main() {
val x =2
when    (x) {
                0, 1   -> print("x    == 0   ou  x   == 1")
                else   -> print("x    tem outro   valor")
}

}
```

```
 x        tem      outro    valor
```

```
fun main() {
val x =0
when    (x) {
                0, 1   -> print("x    == 0   ou  x   == 1")
                else   -> print("x    tem outro   valor")
}

}
```

```
 x        ==       0        ou       x        ==       1
```

```
fun main() {
val x =1
when    (x) {
                0, 1   -> print("x    == 0   ou  x   == 1")
                else   -> print("x    tem outro   valor")
}

}
```

```
 x        ==       0        ou       x        ==       1
```

# ESTRUTURAS DE DECISÃO

- **Expressão When-Exemplo-04**
- Poderia também verificar se x está dentro de um intervalo de valores. Por exemplo, gostaria de verificar se x está entre 1 e 10:

```
fun main() {
val x =7
when    (x) {
    in  1..10   ->  print("x está no intervalo")
    else    ->  print("x está fora do intervalo")

}

}
```

```
 x       está    no      intervalo
```

```
fun main() {
val x =15
when    (x) {
    in  1..10   ->  print("x está no intervalo")
    else    ->  print("x está fora do intervalo")

}
```

```
 x está fora do intervalo
```

# ESTRUTURAS DE DECISÃO

- **Expressão When-Exemplo-05**

```
fun main() {
    println(whenAssign("Olá"))
    println(whenAssign(3.4))
    println(whenAssign(1))
    println(whenAssign(MyClass()))
}

fun whenAssign(obj: Any): Any {
    val resultado = when (obj) { // 1
        1 -> "Uma"               // 2
        "Olá" -> 1               // 3
        is Long -> false         // 4
        else -> 42               // 5
    }
    return resultado
}

class MyClass
```

```
1
42
Uma
42
```

```kotlin
fun main() {
    println("Hello, world!!!")
    println ("Bem vindos a programação com Kotlin!!!");//usando pronto e virgula
    print ("Boa tarde a todos!!!")
}
```

```
Hello, world!!!
Bem vindos a programação com Kotlin!!!
Boa tarde a todos!!!
```

# ESTRUTURAS DE REPETIÇÃO

- Uma estrutura de repetição é utilizada quando queremos repetir determinado trecho de código. Imagine sua lista de contatos do WhatsApp, ela provavelmente está armazenada em um banco de dados e, quando você abre o aplicativo, a lista é montada na tela para você ter acesso a seus contatos. Este é um bom exemplo de uma estrutura de repetição em ação. Imagine que, para montar um contato na tela, o programador montou um bloco de código, mas esse código precisa ser executado para todos os contatos da lista. Então esse código é colocado dentro de uma estrutura de repetição que itera por todos os contatos da lista e monta cada um na tela!

# ESTRUTURAS DE REPETIÇÃO

- Veremos duas estruturas de repetição em **Kotlin**. A linguagem possui mais, mas estas são as mais utilizadas. São elas o for e o **while** . Vamos ver primeiro o **for** .
- O **for** é amplamente utilizado para iterar listas, sendo adequado quando sabemos o número de interações que o programa precisa fazer. Veja um exemplo:

# ESTRUTURAS DE REPETIÇÃO

- **FOR** : Exemplo
- Nesse exemplo, o **for** itera uma lista com valores inteiros, e esse for é executado em todos os itens da lista, desde o primeiro até o último. A cada interação, ele modifica a variável **i** com o valor daquela posição da lista.

```
fun main() {
    val lista = listOf(1,2,3,4)
for(i in lista) {
                println("Item: $i")
```

```
Item:   1
Item:   2
Item:   3
Item:   4
```

# ESTRUTURAS DE REPETIÇÃO

- Em alguns casos, além do valor, precisamos do índice em que aquele valor está na lista. Para esses casos poderíamos usar o seguinte código:

```
fun main() {
  val   lista   =   listOf(1,2,3,4)
for((indice ,valor) in  lista.withIndex()){
    println("índice:    $indice    valor:  $valor")
```

```
índice: 0              valor:  1
índice: 1              valor:  2
índice: 2              valor:  3
índice: 3              valor:  4
```

# ESTRUTURAS DE REPETIÇÃO

- Outra estrutura de repetição disponível na linguagem é o **while** , cujo funcionamento é diferente do for , pois repete um trecho de código enquanto uma condição não for satisfeita. Por exemplo:
- O código repete o print enquanto a variável x tiver um valor menor que 10.

```
fun main() {
 var     x    =    0
    while    (x   <    10) {
         println(x.toString())
         x++
}
}
```

```
0
1
2
3
4
5
6
7
8
9
```