Curso de Android

Programação em Kotlin

Enzo Santos

Faculdade de Computação
Universidade Federal do Pará

30 de Janeiro de 2020

Sumário

- Variáveis
- Modificadores
- Tipos primitivos
- Impressão e leitura de variáveis
- Operações entre variáveis
- Estruturas de condição
- Estruturas de repetição
- Funções
- Classes

Variáveis

Em Kotlin, uma variável pode ser declarada de três formas:

```
<modificador> <nome>: <tipo>

<modificador> <nome> = <valor>

<modificador> <nome>: <tipo> = <valor>
```

Variáveis

Atribuição por valor

```
<modificador> <nome> = <valor>
```

Essa é a forma mais comum de declaração. Criar uma variável nessa forma atribui a ela um valor e automaticamente o tipo desse valor.

Exemplo de atribuição

```
var idade = 18
```

Variáveis

Atribuição por tipo

```
<modificador> <nome>: <tipo>
```

Nessa forma, a variável não possui um valor definido, apenas é declarada no escopo do programa com um tipo. Para poder utilizar a variável nessa forma de declaração, é preciso fazer uma atribuição a algum valor do mesmo tipo atribuído em alguma parte posterior do código.

Exemplo de atribuição por tipo

```
var idade: Int
```

Variáveis

Atribuição por tipo e valor

```
<modificador> <nome>: <tipo> = <valor>
```

Utilizar essa declaração equivale a usar a declaração por valor, porém é especificado explicitamente o tipo de variável que será utilizada.

Exemplo de atribuição

```
var idade: Int = 18
```

Modificadores

O modificador de uma variável permite que seja definida como a variável poderá ser modificada durante a execução do programa, em termos de leitura e escrita.

Modificador `var`

Permite que a variável seja tanto lida quanto alterada posteriormente.

Modificador `val`

Permite que a variável seja apenas lida e não seja alterada em tempo de execução.

Modificador `const val`

Permite que a variável seja apenas lida e não seja alterada em tempo de compilação.

Tipos primitivos

Nome	Bits	Descrição
`Long`	64	Número inteiro de -2^{64} até 2^{64}.
`Int`	32	Número inteiro de -2^{32} até 2^{32}.
`Short`	16	Número inteiro de -2^{16} até 2^{16}.
`Byte`	8	Número inteiro de -2^{8} até 2^{8}.
`Double`	64	Número real de 2^{-1024} até 2^{1024}.
`Float`	32	Número real de 2^{-128} até 2^{128}.
`Char`	16	Caractere no formato UTF-16.
`String`	–	Sequência de objetos do tipo `Char`.
`Boolean`	1	Pode ser `true` ou `false`.

Operações entre variáveis

Operadores matemáticos

Adição (+), subtração (−), multiplicação (∗), divisão (/) e resto da divisão (%).

Operadores de atribuição

Atribuição (=), atribuição com soma (+=), atribuição com subtração (−=), atribuição com multiplicação (∗=), ...

Operadores lógicos

E (&&), ou (||) e não (!).

Operadores de comparação

Igual (==), diferente (!=), maior (>), menor (<), maior ou igual (>=) e menor ou igual (<=).

Impressão e leitura de variáveis

Podemos imprimir o conteúdo de uma variável na tela por meio de duas funções: `print` e `println`.

Exemplo de impressão de variáveis

```
val nome = "Pedro"
val idade = 19
val peso = 65.5
val msg = "$nome tem $idade anos e pesa $peso kg."
println(msg)
// SAÍDA: Pedro tem 19 anos e pesa 65.5 kg.
```

Impressão e leitura de variáveis

Para ler um valor digitado pelo usuário no console, existe a função `readLine()!!`, que retorna o texto digitado pelo usuário em uma variável do tipo `String`.

Exemplo de leitura de variáveis

```
print("Digite o seu nome: ")
val nome = readLine()!!
println("Bem-vindo, $nome!")
```

Impressão e leitura de variáveis

Tratamento de entrada

Para converter o texto recebido para outro tipo primitivo, é possível utilizar a família de métodos `parse`.

De `String` para	**utiliza-se o método**
`Long`	`Long.parseLong(string)`
`Int`	`Integer.parseInt(string)`
`Short`	`Short.parseShort(string)`
`Byte`	`Byte.parseByte(string)`
`Double`	`Double.parseDouble(string)`
`Float`	`Float.parseFloat(string)`
`Char`	`Character.parseChar(string)`
`Boolean`	`Boolean.parseBoolean(string)`

Estruturas de condição

Uma estrutura de condição é formada por `if`, `else if` e/ou `else`. As condições devem estar entre parênteses e devem retornar um valor do tipo `Boolean`, podendo ser tanto operadores lógicos quanto operadores de comparação.

Caso o código referente a cada condição ocupe mais de uma linha, o mesmo deverá estar entre chaves, caso contrário o uso de chaves é opcional.

Estruturas de condição

Exemplo de uma estrutura de condição

```
val idade = Integer.parseInt(readLine()!!)
var mensagem: String

if (idade < 13){
    mensagem = "Você ainda não pode ver o filme"

} else if (idade < 16){
    mensagem = "Você apenas pode ver " +
            "o filme com os seus pais"

} else
    mensagem = "Você pode ver o filme sozinho"
```

Estruturas de condição

Como uma estrutura de condição no Kotlin funciona como uma expressão (ou seja, pode ser atribuída a uma variável), a variável `mensagem` do exemplo anterior pode ser atribuída diretamente à estrutura de condição.

Utilizando estrutura de condição como expressão

```
val mensagem = if (idade < 13)
        "Você ainda não pode ver o filme"
    else if (idade < 16)
        "Você pode ver o filme com seus pais"
    else
        "Você pode ver o filme sozinho"
```

Estruturas de condição

Uma alternativa para o `if` é a estrutura `when`, que pode ser utilizada quando a condição depende apenas do valor de uma variável. Essa estrutura também pode ser utilizada como uma expressão e ser atribuída a uma variável.

Exemplo da estrutura `when`

```
val idade = Integer.parseInt(readLine()!!)
val mensagem = when (idade){
    in 0..12 -> "Não permitido"
    in 13..15 -> "Com acompanhante"
    else -> "Sozinho"
}
```

Estruturas de repetição

Uma estrutura de repetição repete um bloco de instruções dependendo de uma determinada condição. Em Kotlin, existem duas: `for` e `while`. A estrutura `for` pode agir tanto em vetores quanto em intervalos de números.

Exemplo da estrutura `for`

```
val nomes = arrayOf("João", "Pedro", "Lucas")
for (nome in nomes)
    print("$nome ")
// SAÍDA: João Pedro Lucas
```

Estruturas de repetição

Construindo uma sequência de números

- Utiliza-se a notação `a..b` quando se deseja gerar uma sequência de números de `a` até `b`. Por exemplo, `1..10` gera `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`.
- Utiliza-se a notação `a until b` quando se deseja gerar uma sequência de números de `a` até `b-1`. Por exemplo, `1 until 10` gera `[1, 2, 3, 4, 5, 6, 7, 8, 9]`.
- Para definir um passo para a sequência, utiliza-se a notação `a until b step c`, que gera uma sequência de números de `a` até `b-1` indo de `c` em `c`.
- Para definir uma sequência decrescente, utiliza-se a notação `a downTo b`, que gera uma sequência de números de `a` até `b-1`, onde `a > b`. É possível também combinar a notação de sequência descrescente com a notação de passo.

Estruturas de repetição

Existem várias formas de acessar o conteúdo de um vetor criado, como utilizando seus elementos, seus índices ou seus elementos junto com seus índices.

Acessando elementos

```
val quadrados = Array(5){ it * it }
// gera [0, 1, 4, 9, 16]

for (quadrado in quadrados)
    println("$quadrado ")
// SAÍDA: 0 1 4 9 16
```

Estruturas de repetição

Acessando índices

```
val zeros = Array(5){ 0 }
// gera [0, 0, 0, 0, 0]
for (i in zeros.indices)
    println("$i ")
// SAÍDA: 0 1 2 3 4
```

Acessando índices e elementos

```
val strings = Array(5){ "${it + 1}" }
// gera ["1", "2", "3", "4", "5"]
for ((i, string) in strings.withIndex())
    println("$i: $string; ")
// SAÍDA: 0: 1; 1: 2; 2: 3; 3: 4; 4: 5;
```

Estruturas de repetição

Além da estrutura `for`, que interage sobre sequências de elementos, existem também estruturas que dependem de condições. A estrutura de repetição `while` primeiro verifica uma condição e, enquanto essa condição for verdade, um bloco de instruções é executado. Já a estrutura `do while` primeiro executa um bloco de instruções e, enquanto uma condição determinada for verdade, esse bloco de instruções é executado.

Estrutura `while`

```
var x = 0
while (x < 10){
    println(x)
    x++
}
```

Estrutura `do while`

```
var x = 0
do {
    println(x)
    x++
} while (x < 10)
```

Exercícios

- Faça um algoritmo que leia um número inteiro e mostre uma mensagem indicando se este número é par ou ímpar e se é positivo, negativo ou neutro.
- Escreva uma sequência de números inteiros decrescentes entre `a` e `b`, onde `a` e `b` devem ser valores passados pelo usuário.
- Escreva um algoritmo que leia dois números inteiros, `a` e `b`, calcule os quadrados e cubos dos números de `a` e `b` e imprima os valores resultantes no formato de tabela, com colunas `Número`, `Quadrado` e `Cubo`, separadas por espaços.

Exercícios

- Elabore um algoritmo que, dada a idade de um nadador, o classifique em uma das seguintes categorias: **infantil A**, de 5 a 7 anos; **infantil B**, de 8 a 10 anos; **juvenil A**, de 11 a 13 anos; **juvenil B**, de 14 a 17 anos; e **adulto**, maiores de 18 anos. A categoria deverá ser impressa na tela.
- Crie um algoritmo que leia um número do usuário e desenhe o V de vingança, usando asteriscos e underscores (_). Exemplo:

```
*_________*
_*_______*_
__*_____*__
___*___*___
____*_*____
_____*_____
```

Funções

Funções podem ser declaradas da seguinte forma em Kotlin:

```
fun <nome>(<argumentos>: <tipo>): <tipo retorno> {
    <corpo função>
    ...
    return <valor retorno>
}
```

Exemplo de função

```
fun soma(a: Int, b: Int): Int {
    val resSoma = a + b
    return resSoma
}
```

Funções

Caso a função não possua um valor de retorno, ou seja, um procedimento, a sua declaração é a seguinte:

```
fun <nome>(<argumentos>: <tipo>){
    <corpo função>
}
```

Exemplo de procedimento

```
fun mostraNumero(num: Int){
    println("NÚMERO: $num")
}
```

Funções

Para chamar uma função, basta digitar seu nome seguido de seus argumentos entre parênteses e separados por vírgulas. Caso haja valor de retorno, o mesmo pode ser atribuído a uma variável, senão, a função pode apenas ser executada.

Exemplo de chamada de função

```
val a = Integer.parseInt(readLine()!!)
val b = soma(a, 2)
mostraNumero(b)
```

Funções

Uma função também pode ter mais de uma declaração no programa com o mesmo nome, caso seus argumentos sejam diferentes. Essa característica é chamada de sobrecarga. Em tempo de execução, o programa irá chamar a função de acordo com os tipos dos parâmetros de entrada.

Exemplo de sobrecarga de funções

```
fun soma(a: String, b: String): String {
    val numA = Integer.parseInt(a)
    val numB = Integer.parseInt(b)
    val res = numA + numB
    return String.valueOf(res)
}
```

Classes

Kotlin é baseada no conceito de objetos, que podem carregar informações tanto em campos (mais conhecidos como atributos) quanto em código no formato de procedimentos (mais conhecidos como métodos).

Exemplo de uma classe `Pessoa`

```
class Pessoa {
    var nome = "Pedro"
    var idade = 33

    fun cumprimenta(){
       println("Olá, eu sou o $nome!")
    }
}
```

Classes

Para criar uma instância (objeto) dessa classe, basta chamar o nome da classe. Como a classe anterior não possui nenhum construtor (isto é, parâmetros que serão utilizados para criar a classe), então podemos criar uma instância utilizando parênteses vazios. Após criada uma instância de uma classe, também é possível visualizar os seus atributos.

```
val p1 = Pessoa()
println(p1.nome)    // SAÍDA: Pedro
println(p1.idade)   // SAÍDA: 33
p1.cumprimenta()    // SAÍDA: Olá, eu sou o Pedro!
```

Classes

Para criar uma classe com parâmetros pré-definidos, é preciso criar um construtor. O construtor é definido ao lado do nome da classe, como se fosse uma função.

Exemplo de classe `Pessoa` com construtor

```
class Pessoa(nome: String, idade: Int){
    val nome = nome
    val idade = idade
}

val p = Pessoa("João", 29)
print("${p.nome} possui ${p.idade} anos.")
// SAÍDA: "João possui 29 anos."
```

Classes

Caso exista mais de uma maneira de se criar o mesmo objeto, é possível utilizar a característica de sobrecarga para atuar sobre um construtor, utilizando o comando `constructor`.

```
class Pessoa {
    val nome: String
    val idade: Int
    constructor(n: String, idade: Int){
        this.nome = n
        this.idade = idade
    }
    constructor(n: String, sn: String, idade: Int){
        this.nome = "$n $sn"
        this.idade = idade
    }
}
```

Classes

É possível criar um método dentro de uma classe apenas definindo uma função dentro dessa classe.

```
class Pessoa(nome: String, idade: Int){
    val nome = nome
    val idade = idade

    // forma alternativa de declarar funções
    fun cumprimenta() = "Olá, o meu nome é $nome!"

    fun cumprimenta(pessoa: Pessoa): String {
        return "Olá ${pessoa.nome}, " +
            "o meu nome é ${this.nome}!"
    }
}
```

Classes

Um atributo de uma classe também pode ter *getters* e *setters*. Um *getter* retorna um valor relacionado a esse atributo e um *setter* define um valor a esse atributo.

Em linguagens como Java, são utilizados métodos para atuarem como *getters* e *setters*, porém o Kotlin possui uma síntaxe específica para essa funcionalidade.

Modificador para *setters*

Variáveis de apenas-leitura, declaradas com `val`, e constantes, declaradas com `const val`, não podem ter *setters*.

Classes

Exemplo de *getter* e *setter*

```
class Pessoa (nome: String, idade: Int){
    var nome = nome
    var idade = idade
    var nomeComIdade: String
        get() = "$nome $idade"
        set(valor){
            val v = valor.split(" ")
            this.nome = v[0]
            this.idade = Integer.parseInt(v[1])
        }
}
```

Classes

Acessando atributos com *getter* e *setter*

```
val p = Pessoa("Marya", 19)
println(p.nome)              // SAÍDA: Marya
println(p.idade)             // SAÍDA: 19
println(p.nomeComIdade)      // SAÍDA: Marya 19

p.nomeComIdade = "Maria 20"
println(p.nome)              // SAÍDA: Maria
println(p.idade)             // SAÍDA: 20
println(p.nomeComIdade)      // SAÍDA: Maria 20
```

Exercícios

Faça um programa utilizando funções e classes que:

- Possua uma classe chamada `Ponto`, com os atributos `x` e `y`.
- Possua uma classe chamada `Retangulo`, com os atributos `largura` e `altura`.
- Possua um método para imprimir os atributos da classe `Ponto`.
- Cada objeto da classe `Retangulo`, ao ser criado, deve ter um vértice de partida, o vértice inferior esquerdo do retângulo, que deve ser um objeto da classe `Ponto`.
- Possua um método para encontrar o centro de um objeto da classe `Retangulo`, que deverá retornar o valor para um objeto do tipo `Ponto`.
- Crie um menu para alterar os valores do retângulo e imprimir o centro deste retângulo.