

Desenvolvimento de Aplicativos

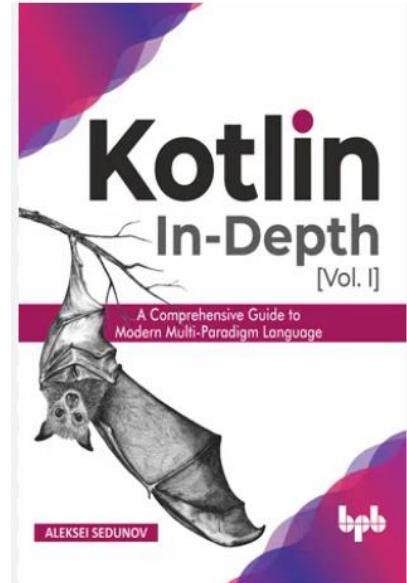
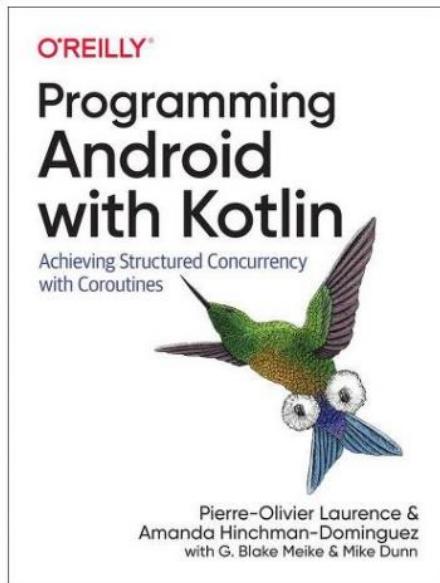
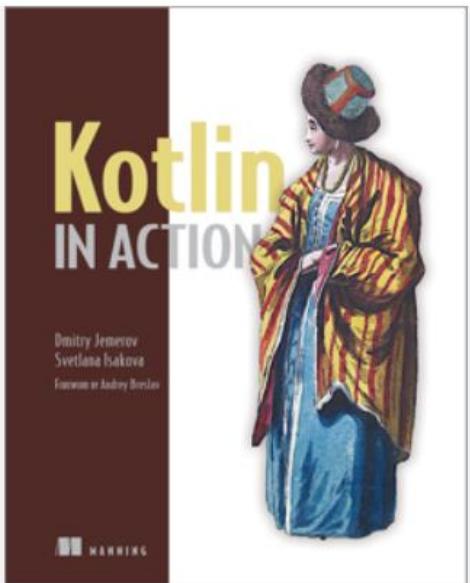
Dispositivos Móveis – Programação de Aplicações para Clientes Móveis

Unidade 5 – Programação Orientada a Objetos com Kotlin



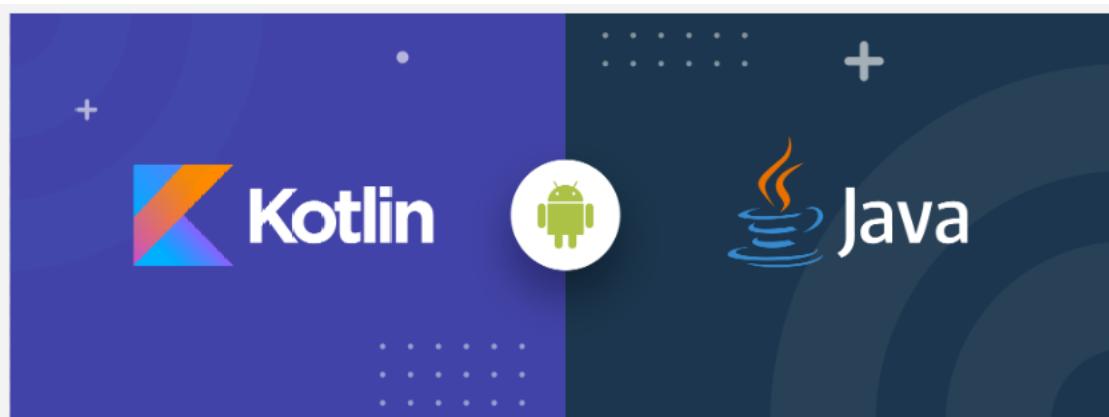
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecido.freitas@online.uscs.edu.br
aparecidovfreitas@gmail.com

Bibliografia



Introdução

- ⊕ **Kotlin** é uma linguagem orientada a objetos e baseada em **classes**;
- ⊕ Ela utiliza **interfaces** e **classes** para definir **tipos personalizados**;
- ⊕ Se você vem do **Java** (ou de qualquer outra linguagem semelhante), se sentirá familiarizado com a forma como **Kotlin** lida com tipos de dados, pois eles são muito semelhantes.



Tipos básicos

- ⊕ **Kotlin** disponibiliza **tipos básicos de dados** os quais são usados para se representar valores comuns em programação;
- ⊕ Estes tipos básicos incluem:

1. Números:

- Inteiros: `Byte`, `Short`, `Int`, `Long`
- Ponto flutuante: `Float`, `Double`

2. **Booleano:** `Boolean` (representa valores verdadeiros/falsos)
3. **Caracteres:** `Char` (representa caracteres)
4. **Strings:** `String` (representa sequências de caracteres)



Como definir tipos personalizados em Kotlin, como por exemplo um **aluno** que tem nome, CPF, email ?

Tipos Personalizados

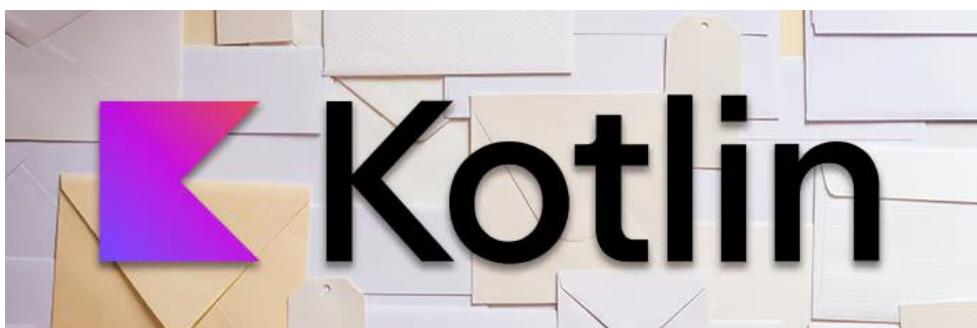
- ⊕ Para criar um tipo personalizado em **Kotlin**, como um **Aluno** que tem nome, **CPF**, **e-mail**, etc., utiliza-se o conceito de **classe**;

- ⊕ Em **OOP**, uma **classe** representa um **modelo** para se criar objetos. Ela define **propriedades** (dados/atributos) e **métodos** (ações/funções) que os objetos criados a partir dela terão.

- ⊕ Em suma, uma classe é como um **molde** a partir do qual instâncias individuais (**objetos**) são criadas, cada uma com suas próprias cópias de dados e capazes de realizar os comportamentos definidos na classe.

Classes

- ⊕ Permitem a criação de **tipos de dados personalizados**;
- ⊕ Agrupam **propriedades (dados)** e **funções (comportamentos)** relacionados em uma única unidade, promovendo a organização e a modularidade do código;
- ⊕ As instâncias de uma classe são **objetos** que mantêm **estado** e **comportamento** definidos pela classe.



Como criar classes em Kotlin ?



```
class Pessoa {  
    private var nome: String  
    private var idade: Int  
  
    constructor(nome: String, idade: Int) {  
        this.nome = nome  
        this.idade = idade  
    }  
  
    fun getNome(): String {  
        return nome  
    }  
  
    fun setNome(nome: String) {  
        this.nome = nome  
    }  
  
    fun getIdade(): Int {  
        return idade  
    }  
  
    fun setIdade(idade: Int) {  
        this.idade = idade  
    }  
}
```

Classe em Kotlin

Classe em Kotlin

```
fun main() {  
    // Criando um objeto da classe Pessoa  
    val pessoa = Pessoa("João", 25)  
  
    // Exibindo os valores iniciais  
    println("Nome: ${pessoa.getNome()}")  
    println("Idade: ${pessoa.getIdade()}")  
  
    // Modificando os valores usando setters  
    pessoa.setNome("Maria")  
    pessoa.setIdade(30)  
  
    // Exibindo os valores atualizados  
    println("Novo Nome: ${pessoa.getNome()}")  
    println("Nova Idade: ${pessoa.getIdade()}")  
}
```

Nome: João
Idade: 25
Novo Nome: Maria
Nova Idade: 30

Classe em Kotlin

```
class Pessoa {  
    private var nome: String  
    private var idade: Int
```

A classe `Pessoa` tem dois atributos:

- `nome` : o nome da pessoa.
- `idade` : a idade da pessoa.

Esses dois atributos são definidos como **privados** (`private`), o que significa que só podem ser acessados e modificados **de dentro da própria classe**, e não diretamente por quem usa a classe.

Classe em Kotlin

```
constructor(nome: String, idade: Int) {  
    this.nome = nome  
    this.idade = idade  
}
```

Construtor: A classe tem um **construtor**, que é uma "função especial" usada para criar um objeto da classe. Ele recebe dois parâmetros, `nome` e `idade`, e os usa para definir os valores iniciais de `nome` e `idade` do objeto.

Classe em Kotlin

```
fun getName(): String {  
    return nome  
}  
  
fun setName(nome: String) {  
    this.nome = nome  
}  
  
fun getIdade(): Int {  
    return idade  
}  
  
fun setIdade(idade: Int) {  
    this.idade = idade  
}
```

Getters e Setters:

- O **getter** é uma função que retorna o valor de um atributo (por exemplo, `getName()` retorna o nome da pessoa).
- O **setter** é uma função que permite modificar o valor de um atributo (por exemplo, `setName(nome: String)` altera o nome da pessoa para o valor passado como argumento).

Tarefa 5.01



Qual o bônus acumulado após 3 anos ?

```
class Funcionario {  
    private var nome: String  
    private var salario: Double  
  
    constructor(nome: String, salario: Double) {  
        this.nome = nome  
        this.salario = salario  
    }  
    fun getNome(): String {  
        return nome  
    }  
    fun calcBonus(): Double {  
        var bonusAcumulado = 0.0  
        var porcentagemBonus = 0.05  
        for (ano in 1..3) {  
            val bonusAnual = salario * porcentagemBonus  
            bonusAcumulado += bonusAnual  
            porcentagemBonus += 0.05  
        }  
        return bonusAcumulado  
    }  
}  
fun main() {  
    val funcionario = Funcionario("Carlos", 2000.0)  
    println("Nome: ${funcionario.getNome()}")  
    println("Bônus Acumulado após 3 anos: ${funcionario.calcBonus()}")  
}
```

- A) 400.0
- B) 600.0
- C) 1000.0
- D) 900.0
- E) 1200.0

Tarefa 5.02



Qual o valor final do investimento após 10 anos ?

```
class Investimento {  
    private var valorInicial: Double  
    private var taxaJuros: Double  
  
    constructor(valorInicial: Double, taxaJuros: Double) {  
        this.valorInicial = valorInicial  
        this.taxaJuros = taxaJuros  
    }  
    fun calcValorFinal(): Double {  
        var valorFinal = valorInicial  
        for (ano in 1..10) {  
            valorFinal += valorFinal * (taxaJuros / 100)  
        }  
        return valorFinal  
    }  
}  
fun main() {  
    val investimento = Investimento(1000.0, 10.0)  
    val valorFinal = investimento.calcValorFinal()  
    println("Valor final do investimento após 10 anos: R$ %.2f".format(valorFinal))  
}
```

- A) 2350.87
- B) 2593.74
- C) 2765.98
- D) 2145.76
- E) 2456.87

Tarefa 5.03



Qual o saldo final após 6 meses ?

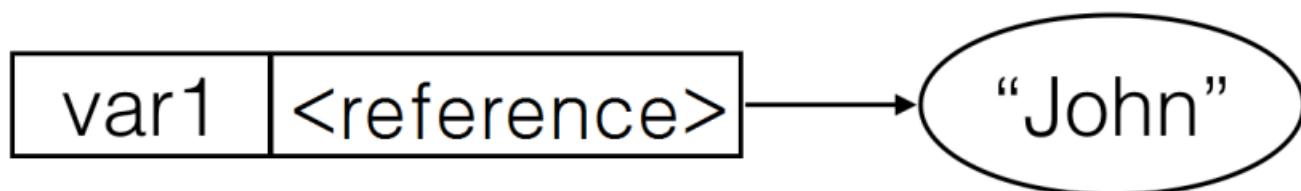
```
class ContaBancaria {  
    private var saldoInicial: Double  
    private var taxaJuros: Double  
    private val depositosMensais = mapOf(  
        1 to 200.0, 2 to 150.0, 3 to 300.0, 4 to 100.0, 5 to 250.0, 6 to 200.0 )  
  
    constructor(saldoInicial: Double, taxaJuros: Double) {  
        this.saldoInicial = saldoInicial  
        this.taxaJuros = taxaJuros  
    }  
    fun calcularSaldoFinal(): Double {  
        var saldoFinal = saldoInicial  
        for ((mes, deposito) in depositosMensais) {  
            saldoFinal += deposito  
            saldoFinal += saldoFinal * (taxaJuros / 100)  
        }  
        return saldoFinal  
    }  
}  
fun main() {  
    val conta = ContaBancaria(1000.0, 2.0)  
    val saldoFinal = conta.calcularSaldoFinal()  
    println("Saldo final após 6 meses: R$ %.2f".format(saldoFinal))  
}
```

- A) 2350.87
- B) 2593.74
- C) 2765.98
- D) 2145.76
- E) 2456.87

Tipos de Referência

- ⊕ Em Kotlin, quando se cria uma instância de uma classe, a variável que se usa não armazena a própria instância, mas sim uma **referência** à um local da memória onde essa instância está armazenada.

```
var var1 = SimplePerson(name = "John")
```

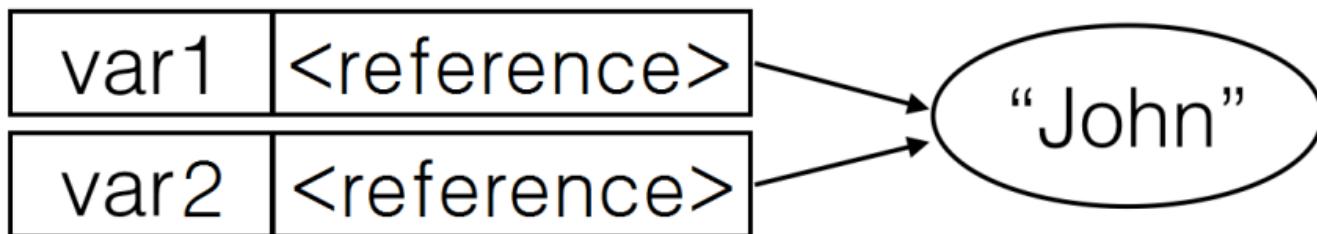


- ⊕ Aqui, **var1** não armazena diretamente o objeto **SimplePerson**. Em vez disso, **var1** contém uma referência ao lugar na memória onde o objeto **SimplePerson("John")** está armazenado.

Tipos de Referência

- ⊕ Pode-se criar uma nova variável **var2** e atribuí-la ao valor de **var1**:

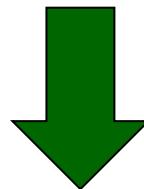
```
var var2 = var1
```



- ⊕ Agora, **var1** e **var2** fazem referência ao mesmo local de memória.

Reference Type

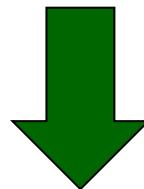
```
class Pessoa {  
    private var nome: String  
  
    constructor(nome: String) {  
        this.nome = nome  
    }  
    fun getNome(): String {  
        return this.nome  
    }  
}  
  
fun main() {  
    var refPessoa1 = Pessoa(nome = "Aparecido")  
    println(refPessoa1.getNome())  
  
    refPessoa1 = Pessoa(nome = "Freitas")  
    println(refPessoa1.getNome())  
  
    var refPessoa2 = refPessoa1;  
    println(refPessoa2.getNome())  
}
```



Em **Kotlin**, quando se cria uma instância de uma classe, a variável que se usa não armazena a própria instância, mas sim uma **referência** à um local da memória onde essa instância está armazenada.

Reference Type

```
class Pessoa {  
    private var nome: String  
  
    constructor(nome: String) {  
        this.nome = nome  
    }  
    fun getNome(): String {  
        return this.nome  
    }  
}  
  
fun main() {  
    var refPessoa1 = Pessoa(nome = "Aparecido")  
    println(refPessoa1.getNome())  
  
    refPessoa1 = Pessoa(nome = "Freitas")  
    println(refPessoa1.getNome())  
  
    var refPessoa2 = refPessoa1;  
    println(refPessoa2.getNome())  
}
```



Em **Kotlin**, quando se cria uma instância de uma classe, a variável que se usa não armazena a própria instância, mas sim uma **referência** à um local da memória onde essa instância está armazenada.

Tarefa 5.04



```

class Produto {
    private var nome: String
    private var preco: Double

    constructor(nome: String, preco: Double) {
        this.nome = nome
        this.preco = preco
    }

    fun getNome(): String {
        return nome
    }

    fun setNome(nome: String) {
        this.nome = nome
    }

    fun getPreco(): Double {
        return preco
    }

    fun setPreco(preco: Double) {
        this.preco = preco
    }

    fun aplicarDesconto(percentual: Double) {
        preco -= preco * (percentual / 100)
    }

    fun calcularTotal(itens: List<Produto>): Double {
        var total = 0.0
        for (item in itens) {
            total += item.getPreco()
        }
        return total
    }
}
  
```

Qual o valor total calculado e impresso na console?

```

fun main() {
    val produto1 = Produto("Caderno", 20.0)
    val produto2 = produto1
    produto2.aplicarDesconto(10.0)
    val itens = listOf(produto1, Produto("Caneta", 5.0), Produto("Lápis", 3.0))
    val total = produto1.calcularTotal(itens) + produto2.calcularTotal(itens)
    println("Total: R$ %.2f".format(total))
}
  
```

- A) 46.00
- B) 48.00
- C) 50.00
- D) 52.00
- E) 54.00

Definindo classes (concisa)

- ⊕ **Kotlin** permite que se defina as propriedades da classe (como nome) diretamente no **construtor primário**;
- ⊕ Isso elimina a necessidade de se escrever campos privados e métodos getter e setter manualmente;
- ⊕ O construtor da classe é declarado dentro dos parênteses logo após o nome da classe.

```
class Pessoa(var nome: String)
```

Exemplo

```
class Pessoa(var nome: String, var idade: Int) {  
  
    fun imprimeIdade() {  
        println("Idade: $idade anos")  
    }  
    fun imprimeNome() {  
        println("Nome: $nome")  
    }  
}  
  
fun main() {  
    val pessoa = Pessoa("Aparecido", 27)  
    pessoa.imprimeNome()  
    pessoa.imprimeIdade()  
}
```

Nome: Aparecido
Idade: 27 anos

Tarefa 5.05



Qual o bônus acumulado após 5 anos ?

```

class Empregado(var nome: String, var salario: Double) {

    fun calcBonus(): Double {
        var bonusAcumulado = 0.0
        var porcentagemBonus = 0.05
        for (ano in 1..5) {
            val bonusAnual = salario * porcentagemBonus
            bonusAcumulado += bonusAnual
            porcentagemBonus += 0.05
        }
        return bonusAcumulado
    }
}

fun main() {
    val empregado = Empregado("Carlos", 2000.0)
    println("Nome: ${empregado.nome}")
    println("Bônus Acumulado após 5 anos: R$ ${empregado.calcBonus()}")
}
  
```

- A) 1200.0
- B) 1300.0
- C) 1400.0
- D) 1500.0
- E) 1600.0

Tarefa 5.06



Qual o valor final do investimento após 10 anos ?

```

class Invest(var valorInicial: Double, var taxaJuros: Double) {

    fun calcValorFinal(): Double {
        var valorFinal = valorInicial
        for (ano in 1..10) {
            valorFinal += valorFinal * (taxaJuros / 100)
        }
        return valorFinal
    }
}

fun main() {
    val investimento = Invest(1000.0, 10.0)
    val valorFinal = investimento.calcValorFinal()
    println("Valor final do investimento após 10 anos: R$ %.2f".format(valorFinal))
}
  
```

A) 2350.87
 B) 2593.74
 C) 2765.98
 D) 2145.76
 E) 2456.87

Tarefa 5.07



Qual o saldo final após 6 meses ?

```

class ContaCorrente(private var saldoInicial: Double, private var taxajuros: Double) {

    private val depositosMensais = mapOf(
        1 to 200.0, 2 to 150.0, 3 to 300.0, 4 to 100.0, 5 to 250.0, 6 to 200.0
    )

    fun calcularSaldoFinal(): Double {
        var saldoFinal = saldoInicial
        for ((_, deposito) in depositosMensais) {
            saldoFinal += deposito
            saldoFinal += saldoFinal * (taxajuros / 100)
        }
        return saldoFinal
    }
}

fun main() {
    val conta = ContaCorrente(5000.0, 2.0)
    val saldoFinal = conta.calcularSaldoFinal()
    println("Saldo final após 6 meses: R$ %.2f".format(saldoFinal))
}
  
```

A) 6350.87
 B) 6593.74
 C) 6765.98
 D) 6916.61
 E) 6456.87

Classes de Dados

- ⊕ Em **Kotlin**, classes de dados, conhecidas como **data classes**, são um recurso especial da linguagem projetado para armazenar dados de maneira concisa e eficiente;
- ⊕ Elas são ideais para classes cujo propósito principal é manter **dados** e oferecem **várias funcionalidades** adicionais para facilitar o manuseio desses dados.

```
data class Usuario(val nome: String, val idade: Int)
```

Classes de Dados

- ⊕ A declaração de uma classe de dados é bastante direta;
- ⊕ Utiliza-se a palavra-chave **data** antes de **class** e definem-se os campos que a classe vai armazenar dentro dos parênteses de construção.

```
data class Usuario(val nome: String, val idade: Int)
```

Classes de Dados

```
data class Usuario(val nome: String, val idade: Int)
```

- ⊕ Neste exemplo, **Usuario** é uma classe de dados com dois campos: **nome** e **idade**;
- ⊕ **Kotlin** automaticamente gera as funções **equals()**, **hashCode()**, **toString()**, e **copy()**, tornando essa classe pronta para ser utilizada em coleções ou quando se precisar de uma maneira fácil e rápida de armazenar coleções de dados.

Classes de Dados

Características Principais

1. **Declaração Simples:** Para declarar uma classe de dados, você usa a palavra-chave `data` antes da declaração da classe. A classe deve ter pelo menos um parâmetro no construtor primário.

```
data class Usuario(val nome: String, val idade: Int)
```

2. **Implementação Automática de Métodos Comuns:** Kotlin automaticamente deriva a implementação de vários métodos úteis, como `equals()`, `hashCode()`, e `toString()`, baseados nas propriedades declaradas no construtor primário da classe.

Classes de Dados

3. **Desestruturação de Objetos:** As classes de dados permitem a desestruturação em múltiplas variáveis.

```
val usuario = Usuario("Alice", 30)  
val (nome, idade) = usuario
```

Como Funciona a Desestruturação

Quando você desestrutura um objeto, está basicamente decompondo-o em um conjunto de variáveis. Isso é feito usando uma sintaxe especial que cria múltiplas variáveis de uma vez. Cada variável corresponde a um componente do objeto.

Desestruturação de Objetos

Vamos considerar uma classe de dados em Kotlin:

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

Você pode desestruturar um objeto `Pessoa` da seguinte maneira:

```
val pessoa = Pessoa("João", 25)
val (nome, idade) = pessoa
println("Nome: $nome, Idade: $idade")
```

Neste exemplo, `nome` recebe o valor de `pessoa.nome` (equivalente a `pessoa.component1()`), e `idade` recebe o valor de `pessoa.idade` (equivalente a `pessoa.component2()`).

Desestruturação de Objetos

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

fun main() {

    val pessoa = Pessoa("Paulo Silva", 35)
    val (nome,idade) = pessoa
    println ("Nome: $nome")
    println ("Idade: $idade")
}
```

Desestruturação de Objetos – Exemplo

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

fun main() {
    val pessoa = Pessoa("Aparecido", 27)
    val (nome, idade) = pessoa
    println("Nome: $nome")
    println("Idade: $idade")
}
```

Nome: Aparecido
Idade: 27

Classes de Dados

4. **Cópias Facilitadas:** Kotlin fornece um método `copy()` para criar uma cópia de um objeto com algumas propriedades alteradas, se necessário.

```
val usuario2 = usuario.copy(nome = "Bob")
```

A função `copy()` em Kotlin é uma funcionalidade especialmente útil em classes de dados (`data classes`). Essa função cria uma nova instância de uma classe de dados, copiando o objeto original, mas também permite modificar algumas das propriedades no processo de cópia. É uma maneira eficiente de criar um novo objeto com a maior parte dos atributos iguais ao objeto original, mas com algumas diferenças.

copy()

A função `copy()` é gerada automaticamente para todas as classes de dados em Kotlin. Quando você chama `copy()`, você pode passar argumentos para algumas ou todas as propriedades da classe de dados. As propriedades para as quais você não passa um argumento serão copiadas do objeto original.

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

```
val pessoaOriginal = Pessoa("Alice", 30)

// Criar uma cópia com a mesma idade, mas com um nome diferente
val novaPessoa = pessoaOriginal.copy(nome = "Bob")
println(novaPessoa) // Saída: Pessoa(nome=Bob, idade=30)
```

Neste exemplo, `novaPessoa` terá o mesmo valor de `idade` que `pessoaOriginal`, mas o `nome` será "Bob".

copy()

Kotlin_04_6.kt

```
1  data class Pessoa (var nome: String, var idade: Int)
2
3  fun main() {
4      val pessoa1 = Pessoa("Carlos", 25)
5      println("Nome: ${pessoa1.nome}, Idade: ${pessoa1.idade}")
6
7      val pessoa2 = pessoa1.copy(nome = "Daniel")
8      println("Nome: ${pessoa2.nome}, Idade: ${pessoa2.idade}")
9
10     val pessoa3 = pessoa1.copy(idade = 26)
11     println("Nome: ${pessoa3.nome}, Idade: ${pessoa3.idade}")
12
13     val pessoa4 = pessoa1.copy(nome = "Eduardo", idade = 27)
14     println("Nome: ${pessoa4.nome}, Idade: ${pessoa4.idade}")
15 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS"
Kotlin_04_6.jar && java -jar Kotl
Nome: Carlos, Idade: 25
Nome: Daniel, Idade: 25
Nome: Carlos, Idade: 26
Nome: Eduardo, Idade: 27
```

copy()

Considerações

- **Imutabilidade:** A função `copy()` é particularmente útil em contextos onde a imutabilidade é importante. Ao usar objetos imutáveis, qualquer modificação resulta em um novo objeto, mantendo o original inalterado.
- **Classes de Dados:** Lembre-se de que `copy()` só está disponível automaticamente em classes de dados.

A função `copy()` é uma maneira prática de trabalhar com objetos imutáveis em Kotlin, permitindo a criação de novas instâncias com variações de um objeto existente de forma concisa e clara.



Construtor Secundário

Kotlin_04_4.kt

```
1  class Circulo(val raio: Double) {  
2  
3      // Construtor secundário  
4      constructor() : this(1.0) {  
5          // "this(1.0)" delega para o  
6          // construtor primário com  
7          // o raio padrão 1.0  
8      }  
9  }  
10  
11  
12 fun main() {  
13     val circulo1 = Circulo();  
14     println(circulo1.raio)  
15  
16     val circulo2 = Circulo(77.5);  
17     println(circulo2.raio)  
18 }
```

[Running] cd "d:\USCS\DISCIPLINAS_U
Kotlin_04_4.jar && java -jar Kotlin
1.0
77.5

Propriedades dos objetos

- ⊕ Representam **características** ou **atributos** que os objetos de uma classe podem ter;
- ⊕ São úteis para modelar objetos de domínio. Tipicamente (em Java), criam-se propriedades definindo-se variáveis membros;
- ⊕ Após isso, deve-se criar **métodos acessores (getter e setter)** para se obter e se definir os valores das variáveis membros;
- ⊕ Em **Kotlin**, isso **não** é necessário pois a linguagem tem suporte nativo para propriedades!



Getter e Setter – Kotlin

- ⊕ Em **Kotlin**, **não** é necessário definir-se **getters** e **setters** manualmente para propriedades simples;
- ⊕ O **compilador** já os gera automaticamente!



- Para `val` (propriedade imutável): O Kotlin gera apenas um **getter**.
- Para `var` (propriedade mutável): O Kotlin gera automaticamente um **getter** e um **setter**.

```
class Pessoa(var nome: String)

fun main() {
    val pessoa = Pessoa("Aparecido Freitas")

    // Usando o getter (acesso à propriedade)
    println(pessoa.nome)

    // Usando o setter (modificando a propriedade)
    pessoa.nome = "João Silva"
    println(pessoa.nome)
}
```

1. **Getter:** Quando você acessa `pessoa.nome`, o getter é chamado para retornar o valor da propriedade.
2. **Setter:** Quando você faz `pessoa.nome = "João Silva"`, o setter é chamado para modificar o valor da propriedade.

Cenário 1

```

// Definindo a classe Pessoa com uma propriedade var (mutável)
class Pessoa(var nome: String) {
    // Neste caso, o Kotlin gera automaticamente os getters e setters
    // - O getter será gerado para acessar o valor da propriedade 'nome'
    // - O setter será gerado para modificar o valor da propriedade 'nome'
}

fun main() {
    // Criando uma instância da classe Pessoa
    var pessoa = Pessoa("Aparecido Freitas")
    // Usando o getter gerado automaticamente para acessar a propriedade 'nome'
    println(pessoa.nome)
    // Neste caso, 'pessoa.nome' utiliza o getter, que foi gerado automaticamente
    // O getter acessa o valor atual de 'nome' e o imprime no console
}

```

Aparecido Freitas

Cenário 2

```
// Definindo a classe Pessoa com uma propriedade var (mutável)
class Pessoa(var nome: String) {
    // Neste caso, o Kotlin gera automaticamente o getter e o setter
    // - O getter será gerado automaticamente para acessar o valor da propriedade 'nome'
    // - O setter será gerado automaticamente para modificar o valor da propriedade 'nome'
}

fun main() {
    // Criando uma instância da classe Pessoa com o nome "Aparecido Freitas"
    var pessoa = Pessoa("Aparecido Freitas")

    // Usando o getter gerado automaticamente para acessar a propriedade 'nome'
    println(pessoa.nome)
    // Neste caso, 'pessoa.nome' utiliza o getter gerado automaticamente
    // O getter acessa o valor atual de 'nome' e o imprime no console

    // Criando uma nova instância da classe Pessoa com o nome "Renato Carioca"
    pessoa = Pessoa("Renato Carioca")

    // Usando o getter novamente para acessar o novo valor de 'nome'
    println(pessoa.nome)
}
```

Cenário 3

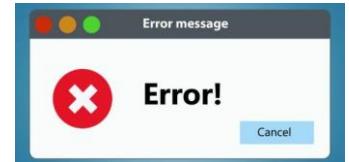
```

// Definindo a classe Pessoa com uma propriedade var (mutável)
class Pessoa(var nome: String) {
    // Neste caso, o Kotlin gera automaticamente o getter e o setter
    // - O getter será gerado automaticamente para acessar o valor da propriedade 'nome'
    // - O setter será gerado automaticamente para modificar o valor da propriedade 'nome'
}

fun main() {
    // Criando uma instância da classe Pessoa com o nome "Aparecido Freitas"
    val pessoa = Pessoa("Aparecido Freitas")

    // Usando o getter gerado automaticamente para acessar a propriedade 'nome'
    println(pessoa.nome)
    // Neste caso, 'pessoa.nome' utiliza o getter gerado automaticamente
    // O getter acessa o valor atual de 'nome' e o imprime no console

    pessoa = Pessoa("Renato Carioca")
    // ERRO: Não é possível atribuir uma nova instância a uma variável 'val'
    // A variável 'pessoa' é uma constante, e você não pode modificar sua referência
    // Se 'pessoa' fosse 'var', você poderia atribuir uma nova instância.
    // O código abaixo causaria um erro de compilação:
}
  
```



Cenário 4

```

// Definindo a classe Pessoa com uma propriedade val (imutável)
class Pessoa(val nome: String) {
    // Neste caso, o Kotlin gera automaticamente apenas o getter
    // - O getter será gerado automaticamente para acessar o valor da propriedade 'nome'
    // - Como 'nome' é val (imutável), NÃO será gerado um setter
}

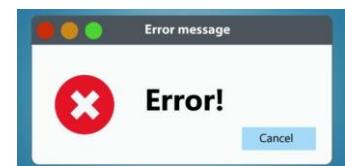
fun main() {
    // Criando uma instância da classe Pessoa com o nome "Aparecido Freitas"
    var pessoa = Pessoa("Aparecido Freitas")

    // Usando o getter gerado automaticamente para acessar a propriedade 'nome'
    println(pessoa.nome)

    // Neste caso, 'pessoa.nome' utiliza o getter gerado automaticamente
    // O getter acessa o valor atual de 'nome' e o imprime no console

    pessoa.nome = "Renato Carioca"
    // ERRO: Não é possível modificar o valor de 'nome', pois ele foi declarado como 'val'
}

```



Cenário 5

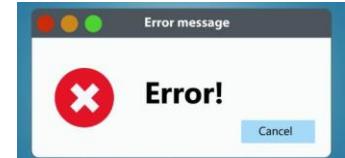
```

// Definindo a classe Pessoa com uma propriedade val (imutável)
class Pessoa(val nome: String) {
    // Neste caso, o Kotlin gera automaticamente apenas o getter
    // - O getter será gerado automaticamente para acessar o valor da propriedade 'nome'
    // - Como 'nome' é val (imutável), NÃO será gerado um setter
}

fun main() {
    // Criando uma instância da classe Pessoa com o nome "Aparecido Freitas"
    val pessoa = Pessoa("Aparecido Freitas")

    // Usando o getter gerado automaticamente para acessar a propriedade 'nome'
    println(pessoa.nome)
    // Neste caso, 'pessoa.nome' utiliza o getter gerado automaticamente
    // O getter acessa o valor atual de 'nome' e o imprime no console

    pessoa = Pessoa("Renato Carioca")
    // ERRO: Não é possível reatribuir a variável 'pessoa', pois ela foi declarada como 'val'
    // Uma variável declarada como 'val' não pode ser modificada para apontar para um novo objeto.
}
  
```



Exemplo

◀ Kotlin_04_5.kt

```
1 class Pessoa(val nome: String)
2
3 fun main() {
4     val pessoa = Pessoa("Aparecido Freitas")
5     println(pessoa.nome)
6 }
```

```
[Running] cd "d:\USCS\DISCI"
Kotlin_04_5.jar && java -ja
Aparecido Freitas
```



No exemplo anterior,
como alterar o valor da
propriedade nome?

Exemplo

◀ Kotlin_04_5.kt

```
1  class Pessoa(val nome: String)
2
3  fun main() {
4      val pessoa = Pessoa("Aparecido Freitas")
5      pessoa.nome = "Renato Carioca"
6      println(pessoa.nome)
7  }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des_APLICATIVOS_Kotlin"
Kotlin_04_5.jar && java -jar Kotlin_04_5.jar
Kotlin_04_5.kt:5:5: error: val cannot be reassigned
    pessoa.nome = "Renato Carioca"
    ^

```



ERROR



Qual a causa do erro ?

Exemplo

Kotlin_04_5.kt

```
1  class Pessoa(val nome: String)
2
3  fun main() {
4      val pessoa = Pessoa("Aparecido Freitas")
5      pessoa.nome = "Renato Carioca"
6      println(pessoa.nome)
7 }
```



```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des_Aplicativ<...>
Kotlin_04_5.jar && java -jar Kotlin_04_5.jar
Kotlin_04_5.kt:5:5: error: val cannot be reassigned
    pessoa.nome = "Renato Carioca"
    ^
```

Exemplo

◀ Kotlin_04_5.kt

```
1  class Pessoa(var nome: String)
2
3  fun main() {
4      var pessoa = Pessoa("Aparecido Freitas")
5      pessoa.nome = "Renato Carioca"
6      println(pessoa.nome)
7  }
```

```
[Running] cd "d:\USCS\DISCI"
Kotlin_04_5.jar && java -ja
Renato Carioca
```

Modificadores de visibilidade

- ⊕ **Kotlin** usa as mesmas **palavras-chave** que **Java** para controlar a visibilidade;
- ⊕ As palavras-chave **public**, **private** e **protected** têm o mesmo significado tanto em **Kotlin** como em **Java**;
- ⊕ No entanto, a diferença está na visibilidade padrão. Em **Kotlin**, a visibilidade padrão é **pública** sempre que se omitir o modificador de visibilidade.

Modificadores de visibilidade

- ⊕ Em **Kotlin**, os modificadores de visibilidade determinam a visibilidade de classes, objetos, interfaces, construtores, funções e propriedades;
- ⊕ Existem quatro modificadores de visibilidade em **Kotlin**:

1. **public (padrão)**: Se nenhum modificador de visibilidade for especificado, `public` é o padrão. Um membro declarado como `public` é visível em qualquer lugar.

kotlin

 Copy code

```
public var x: Int = 0 // Visível em qualquer lugar
```

Modificadores de visibilidade

2. **private**: Quando um membro é marcado como `private`, ele é visível apenas dentro da classe ou do arquivo em que foi declarado. No caso de classes de nível superior (que não estão dentro de outras classes), `private` significa visibilidade no arquivo.

kotlin

 Copy code

```
private var y: Int = 1 // Visível apenas dentro da classe ou arquivo
```

Modificadores de visibilidade

3. **protected**: Este modificador é semelhante ao `private`, mas também permite a visibilidade em subclasses. `protected` não é aplicável a classes de nível superior.

kotlin

 Copy code

```
open class A {  
    protected var z: Int = 2 // Visível na classe A e suas subclasses  
}
```

Modificadores de visibilidade

4. **internal**: Um membro marcado como `internal` é visível em todo o módulo em que está declarado. Um módulo é um conjunto de arquivos Kotlin compilados juntos, como um projeto IntelliJ IDEA, um projeto Maven ou Gradle, etc.

kotlin

 Copy code

```
internal var w: Int = 3 // Visível em todo o módulo
```

Modificadores de visibilidade

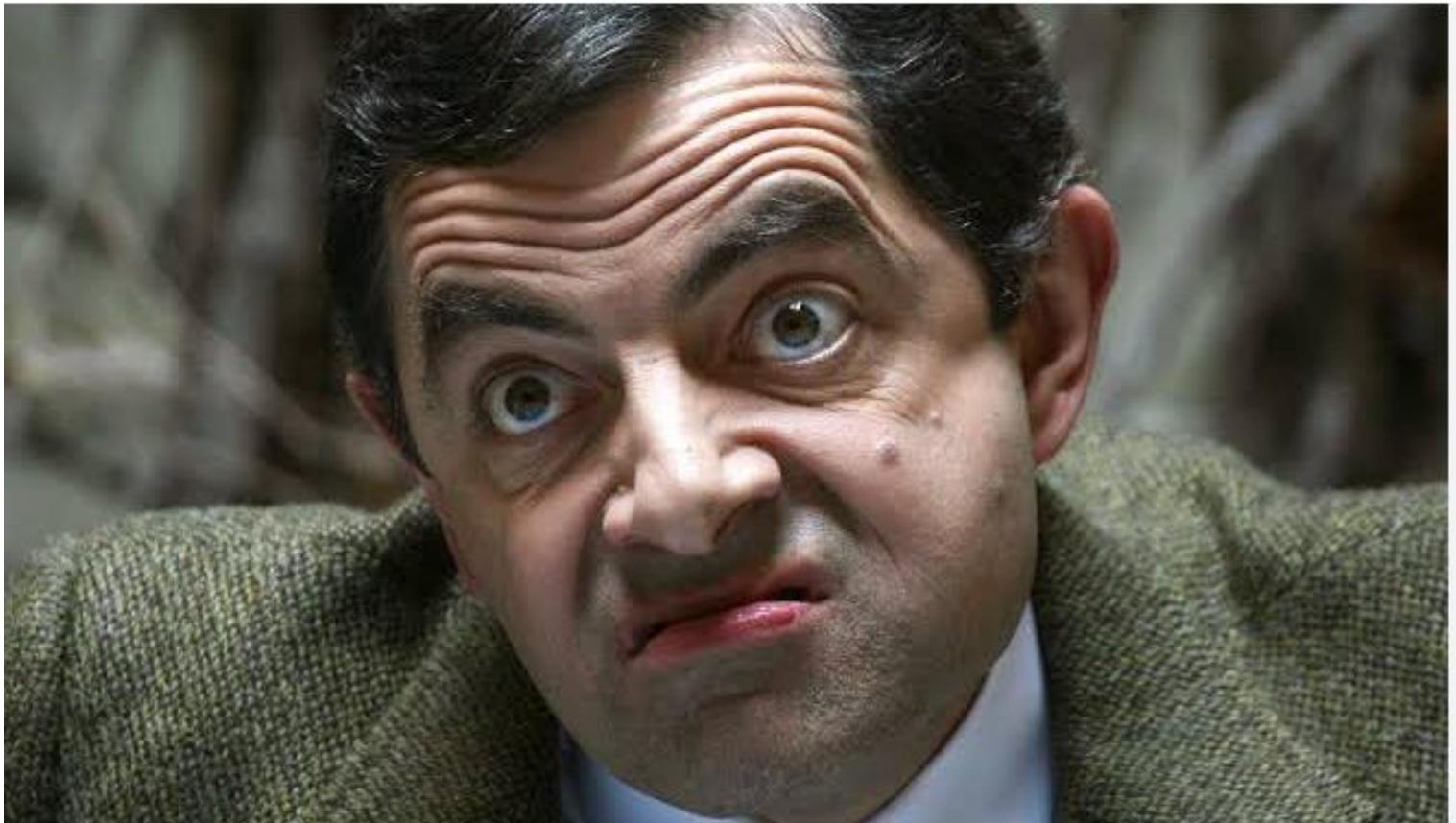
```
class Visibilidade {  
    var A: Int = 1 // Pode ser acessada de qualquer lugar  
    private var B: Int = 2 // Só pode ser acessada dentro desta classe  
  
    fun mostrarValores() {  
        println("Publico: $A, Privado: $B")  
    }  
}
```

Modificadores de visibilidade

```
fun main() {  
    val trab = Visibilidade()  
  
    // Acesso permitido: propriedade A é pública  
    println(trab.A)  
  
    // Acesso negado: propriedade B é privada,  
    // logo a linha abaixo geraria um erro se descomentada  
    // println(trab.B)  
  
    // Método público que acessa a propriedade privada internamente  
    trab.mostrarValores()  
}
```



Kotlin_04_9.kt:18:18: error: cannot access 'B': it is private in 'Visibilidade'
 println(trab.B)
 ^



Como acessar o dado `private` fora
da classe em que foi declarado?

Como acessar o dado private fora da classe em que foi declarado?

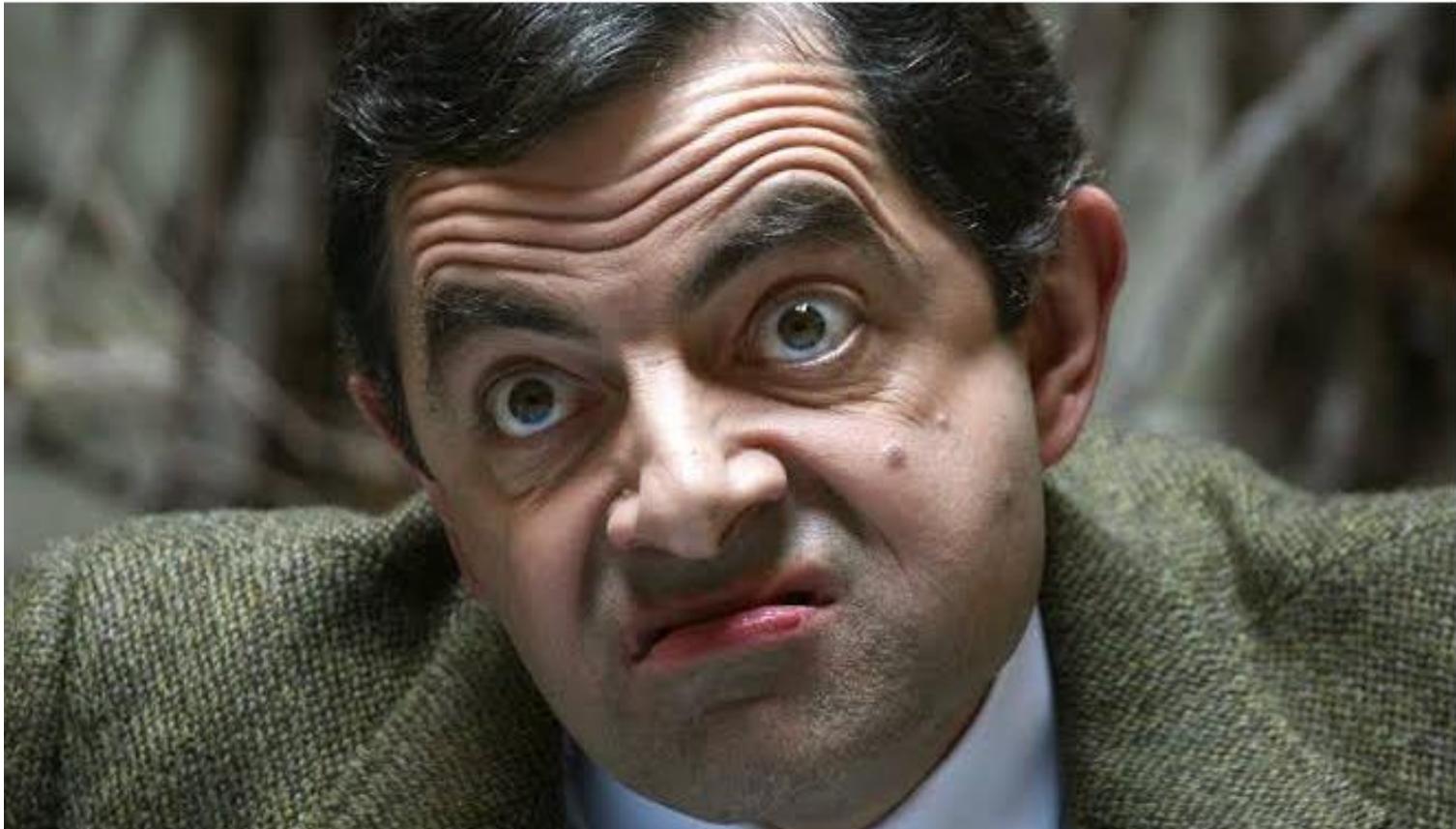
```
class Visibilidade {  
    var A: Int = 1 // Pode ser acessada de qualquer lugar  
    private var B: Int = 2 // Só pode ser acessada dentro desta classe  
  
    fun mostrarValores() {  
        println("Publico: $A, Privado: $B")  
    }  
  
    // Método público para obter o valor de B  
    fun getB(): Int {  
        return B  
    }  
}
```

Como acessar o dado private fora da classe em que foi declarado?

```
fun main() {  
    val trab = Visibilidade()  
  
    // Acesso permitido: propriedade A é pública  
    println(trab.A)  
  
    // Agora é possível acessar o valor de B  
    // indiretamente através do método público getB()  
    println(trab.getB())  
  
    // Método público que acessa a propriedade privada internamente  
    trab.mostrarValores()  
}
```



```
f ($?) { java -jar Kotlin_04_10.jar }  
1  
2  
Publico: 1, Privado: 2
```



Como alterar o dado `private` fora da classe em que foi declarado?

Como alterar o dado private fora da classe em que foi declarado?

```
class Visibilidade {  
    var A: Int = 1 // Pode ser acessada de qualquer lugar  
    private var B: Int = 2 // Só pode ser acessada dentro desta classe  
  
    fun mostrarValores() {  
        println("Publico: $A, Privado: $B")  
    }  
  
    // Método público para obter o valor de B  
    fun getB(): Int {  
        return B  
    }  
  
    // Método público para alterar o valor de B  
    fun setB(value: Int) {  
        B = value  
    }  
}
```

Como alterar o dado private fora da classe em que foi declarado?

```
fun main() {  
    val trab = Visibilidade()  
  
    // Acesso permitido: propriedade A é pública  
    println(trab.A)  
  
    // Alterar o valor de B usando o método setB  
    trab.setB(5)  
  
    // Verificar a alteração acessando B  
    // indiretamente através do método getB  
    println(trab.getB())  
  
    // Método público que acessa a propriedade  
    // privada internamente  
    trab.mostrarValores()  
}  
f ($?) { java -jar Kotlin_04_11.jar }  
1  
5  
Publico: 1, Privado: 5
```

Bloco init

- ⊕ O **init** é um bloco de código que é executado quando um objeto de uma classe é criado;
- ⊕ Ele pode ser usado para **inicializar propriedades**, executar tarefas de **inicialização** ou algum processamento que se queira fazer antes que o objeto esteja totalmente inicializado;
- ⊕ O **init** não é obrigatório, mas pode ser útil em muitos casos;
- ⊕ Escrito dentro do corpo da classe e é usado para complementar a lógica de inicialização do construtor primário.

Bloco init - Exemplo

```

class Pessoa(var nome: String?, var idade: Int) {

    init {
        // Inicializa a propriedade `nome`
        nome = "Ana Maria"

        // Executa uma tarefa de inicialização
        println("Criando um novo objeto da classe Pessoa")
    }

}

fun main() {
    val pessoa = Pessoa(null,23)
    println(pessoa.nome)
    println(pessoa.idade)

    pessoa.nome = "Pedro"
    pessoa.idade = 55
    println(pessoa.nome)
    println(pessoa.idade)
}
  
```

```

[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des_A"
Kotlin_04_2.jar && java -jar Kotlin_04_2.jar
Criando um novo objeto da classe Pessoa
Ana Maria
23
Pedro
55
  
```

Bloco init - Exemplo

◀ Kotlin_04_3.kt

```
1  class Pessoa( _nome: String, _idade: Int) {  
2  
3      var nome: String  
4      var idade: Int  
5  
6      init {  
7          // Inicializa a propriedade "nome"  
8          nome = _nome  
9  
10         // Inicializa a propriedade "idade"  
11         idade = _idade  
12  
13         // Executa uma tarefa de inicialização  
14         println("Criando um novo objeto da classe Pessoa")  
15     }  
16 }  
17  
18 fun main() {  
19     val pessoa = Pessoa("Ana Maria",19)  
20     println(pessoa.nome)  
21     println(pessoa.idade)  
22 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des...  
Kotlin_04_3.jar && java -jar Kotlin_04_3.jar  
Criando um novo objeto da classe Pessoa  
Ana Maria  
19
```

Bloco init - Exemplo

◀ Kotlin_04_3.kt

```
1  class Pessoa( _nome: String, _idade: Int) {  
2  
3      var nome: String  
4      var idade: Int  
5  
6      init {  
7          // Inicializa a propriedade "nome"  
8          nome = _nome  
9  
10         // Inicializa a propriedade "idade"  
11         idade = _idade  
12  
13         // Executa uma tarefa de inicialização  
14         println("Criando um novo objeto da classe Pessoa")  
15     }  
16 }  
17  
18 fun main() {  
19     val pessoa = Pessoa("Ana Maria",19)  
20     println(pessoa.nome)  
21     println(pessoa.idade)  
22 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\CC_Des...  
Kotlin_04_3.jar && java -jar Kotlin_04_3.jar  
Criando um novo objeto da classe Pessoa  
Ana Maria  
19
```

Herança

- + A herança em **Kotlin**, assim como em outras linguagens de programação orientadas a objeto, é um conceito fundamental que permite a uma classe **herdar propriedades e funções** de outra classe;
- + **Kotlin** maneja a herança de uma maneira um pouco diferente do **Java**, especialmente em relação à **herança de classes** e à implementação de **interfaces**.



Herança

- ⊕ Por meio da herança em **Kotlin**, pode-se criar uma nova classe baseada em uma classe existente;
- ⊕ A nova **classe**, conhecida como classe **filha** ou **subclasse**, **herda atributos e métodos da classe pai ou superclasse**, permitindo **reutilizar** código comum e promover uma organização lógica do código.



Herança

- ⊕ Imagine que você está desenvolvendo um aplicativo para gerenciar **veículos**;



- ⊕ Você tem classes básicas como **Veiculo**, que possui atributos como marca, modelo e ano, além de métodos como **ligar** e **desligar**;



- ⊕ Agora, se você quiser introduzir diferentes tipos de veículos, como **Carro** e **Moto**, em vez de reescrever toda a lógica comum em cada uma dessas novas classes, você pode fazer com que elas herdem da classe **Veiculo**.

Herança

```
// Definição da superclasse Veiculo
open class Veiculo(val marca: String, val modelo: String, val ano: Int) {
    fun ligar() {
        println("Ligando o veículo $modelo da marca $marca, ano $ano.")
    }

    fun desligar() {
        println("Desligando o veículo $modelo.")
    }
}
```

Herança

```
// Definição da subclasse Carro, que herda de Veiculo
class Carro(marca: String, modelo: String, ano: Int, val numeroDePortas: Int)
    : Veiculo(marca, modelo, ano) {

    fun abrirPortas() {
        println("Abrindo as $numeroDePortas portas do carro $modelo.")
    }
}
```

Herança

```
// Função main para execução
fun main() {
    // Criando uma instância de Carro
    val meuCarro = Carro("Toyota", "Corolla", 2020, 4)

    // Executando métodos herdados
    meuCarro.ligar()
    // Executando método específico da subclasse
    meuCarro.abrirPortas()
    // Continuação da execução de métodos herdados
    meuCarro.desligar()
}
```

```
f ($?) { java -jar Kotlin_04_12.jar }
```

Ligando o veículo Corolla da marca Toyota, ano 2020.

Abrindo as 4 portas do carro Corolla.

Desligando o veículo Corolla.

Herança

- 1. Classes Abertas:** Em Kotlin, todas as classes são `final` por padrão, o que significa que elas não podem ser herdadas a menos que sejam explicitamente marcadas como `open` ou `abstract`. Isso é diferente do Java, onde as classes são herdáveis por padrão.

```
open class Base { // Classe aberta que pode ser herdada
    open fun v() {}
}
```

- 2. Herança de Classes:** Para herdar uma classe em Kotlin, você coloca o nome da classe base após o nome da subclasse, separado por dois pontos.

```
class Derived : Base() { // Herda de Base
}
```

Sobreposição de métodos

1. **Métodos Abertos:** Assim como as classes, os métodos em Kotlin são `final` por padrão. Para permitir que um método seja sobreescrito em uma subclasse, ele deve ser marcado como `open`.

```
open class Base {  
    open fun v() {}  
    fun nv() {} // Este método não pode ser sobreescrito  
}  
  
class Derived() : Base() {  
    override fun v() {} // Sobreposição do método  
}
```

2. **Palavra-chave `override`:** Ao sobreescrver um método, você deve usar a palavra-chave `override`. Isso garante explicitamente que você está sobreescrivendo um método da classe base.

Herança – Exemplo

Classe Base

Primeiro, definimos uma classe base. Lembre-se de que, em Kotlin, as classes são `final` por padrão, então precisamos usar a palavra-chave `open` para permitir que outras classes herdem dela.

```
open class Veiculo(val marca: String) {  
    open fun acelerar() {  
        println("Veículo está acelerando")  
    }  
  
    fun parar() {  
        println("Veículo parou")  
    }  
}
```

Neste exemplo, `Veiculo` é uma classe com uma propriedade `marca` e dois métodos: `acelerar()` e `parar()`. O método `acelerar` é `open`, o que significa que pode ser sobreescrito por subclasses.

Herança – Exemplo

◀ Kotlin_04_7.kt

```
1  open class Veiculo(val marca: String) {
2      open fun acelerar() {
3          println("Veiculo acelerando...")
4      }
5
6      fun parar() {
7          println("Veiculo parou...")
8      }
9  }
10 class Carro(marca: String, val numeroDePortas: Int) : Veiculo(marca) {
11     override fun acelerar() {
12         println("Carro $marca com $numeroDePortas portas acelerando...")
13     }
14 }
```

Herança – Exemplo

```
15 fun main() {  
16     val meuVeiculo = Veiculo("Generico")  
17     meuVeiculo.acelerar() // Saída: Veículo está acelerando  
18     meuVeiculo.parar()   // Saída: Veículo parou  
19  
20     val meuCarro = Carro("Ford", 4)  
21     meuCarro.acelerar() // Saída: Carro Ford com 4 portas acelerando  
22     meuCarro.parar()   // Saída: Veículo parou  
23 }
```

```
[Running] cd "d:\USCS\DISCIPLINAS_USCS\Kotlin_04_7.jar && java -jar Kotlin_04_7  
Veiculo acelerando...  
Veiculo parou...  
Carro Ford com 4 portas acelerando...  
Veiculo parou...
```

Neste exemplo, `meuCarro.acelerar()` chama a versão sobrescrita de `acelerar()`, enquanto `meuCarro.parar()` chama o método `parar()` herdado de `Veiculo`, que não foi sobreescrito em `Carro`.

Herança – outro exemplo

```
// Classe base Pessoa com propriedades básicas
open class Pessoa(val nome: String, val idade: Int) {
    fun identificar() {
        println("Nome: $nome, Idade: $idade")
    }
}

// Classe Estudante herda de Pessoa e adiciona a propriedade curso
class Estudante(nome: String, idade: Int, val curso: String)
    : Pessoa(nome, idade) {
    fun estudar() {
        println("$nome está estudando $curso.")
    }
}
```

Herança – outro exemplo

```
fun main() {  
    val pessoa = Pessoa("Carlos", 40)  
    pessoa.identificar()  
  
    val estudante = Estudante("Ana", 20, "Kotlin")  
    estudante.identificar() // Método herdado da classe Pessoa  
    estudante.estudar() // Método específico da classe Estudante  
}
```

```
f (?) { java -jar Kotlin_04_13.jar }  
Nome: Carlos, Idade: 40  
Nome: Ana, Idade: 20  
Ana está estudando Kotlin.
```

Herança – outro exemplo

Classe `Pessoa`: A classe base contém propriedades comuns `nome` e `idade` e um método `identificar` que imprime essas informações. A classe é marcada como `open`, o que significa que outras classes podem herdar dela.

Classe `Estudante`: A classe `Estudante` herda da classe `Pessoa` e adiciona uma nova propriedade `curso`. Além disso, possui um método próprio `estudar` que imprime uma mensagem indicando que o estudante está estudando seu curso.

Função `main`: Aqui, criamos instâncias de `Pessoa` e `Estudante` e chamamos seus métodos. Nota-se que a instância de `Estudante` pode usar o método `identificar` herdado de `Pessoa`, além de seu método `estudar`.

Declaração de Objetos

- ⊕ A palavra-chave **static** de Java não foi incluída na lista de palavras-chave do **Kotlin**;
- ⊕ Não existe um equivalente direto para **static** em **Kotlin**;
- ⊕ Em seu lugar, o **Kotlin** introduz as palavras-chave **object** e **companion**. A palavra-chave **object** permite definir uma classe e sua instância ao mesmo tempo. Mais especificamente, ela define apenas uma **única instância** dessa classe, o que torna essa palavra-chave adequada para se definir **singletons** em **Kotlin**.



Companion Object

- ⊕ Em uma classe **Kotlin**, pode-se usar **companion object** para definir membros que serão estáticos em termos de **Java**, ou seja, membros que podem ser acessados sem a necessidade de uma instância da classe.

```
class Exemplo {  
    companion object {  
        fun metodoEstatico() {  
            println("Método estático chamado")  
        }  
  
        val propriedadeEstatica = "Propriedade Estática"  
    }  
}  
  
fun main() {  
    // Acessando diretamente através da classe, sem instância  
    Exemplo.metodoEstatico()  
    println(Exemplo.propriedadeEstatica)  
}
```

Object

- ⊕ A palavra-chave **object** é usada em **Kotlin** para se criar um objeto de uma classe anônima ou para se definir um **singleton**, uma classe da qual apenas uma instância existe;

```
object Singleton {  
    fun metodoSingleton() {  
        println("Método do Singleton chamado")  
    }  
  
    val propriedadeSingleton = "Propriedade do Singleton"  
}  
  
fun main() {  
    // Acessando membros do singleton  
    Singleton.metodoSingleton()  
    println(Singleton.propriedadeSingleton)  
}
```

- ⊕ Nesse caso, ‘**Singleton**’ é um objeto único, e seus membros podem ser acessados diretamente.

object e companion object

- Use `companion object` para membros que você quer que sejam associados à classe em si e não a instâncias da classe, similar a membros estáticos em Java.
- Use `object` para criar singletons ou objetos de uma única instância, que são úteis quando você precisa de uma única instância global em toda a sua aplicação.

Ambos `companion object` e `object` são formas eficazes de trabalhar com membros estáticos e singletons em Kotlin, oferecendo uma abordagem mais flexível e segura do que o modelo tradicional `static` do Java.

Interfaces

- ⊕ Interfaces em **Kotlin** representam um conceito fundamental da programação orientada a objetos;
- ⊕ Elas são semelhantes às interfaces em outras linguagens como **Java**, mas com algumas características adicionais oferecidas por **Kotlin**;
- ⊕ Uma interface define um **contrato** que as classes podem implementar. Este contrato é composto por declarações de funções e propriedades que as classes de implementação devem fornecer.

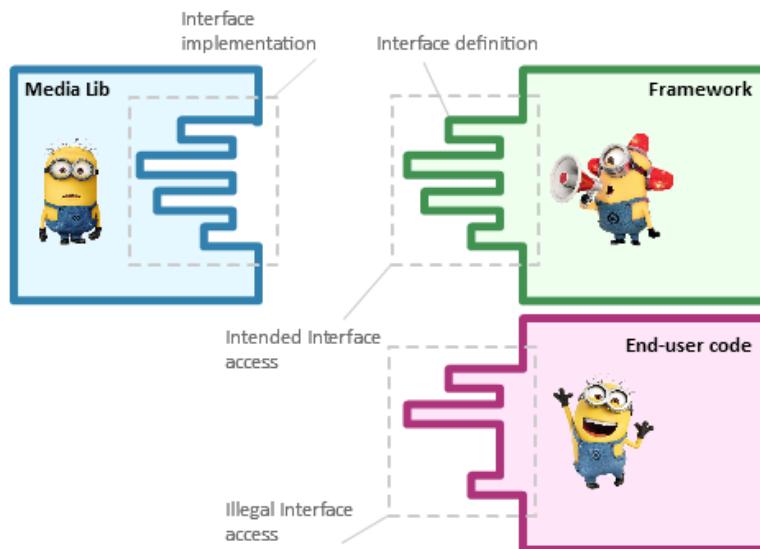


Interfaces

- 1. Métodos Abstratos:** Uma interface pode conter declarações de métodos abstratos (sem corpo). As classes que implementam a interface devem fornecer implementações para esses métodos.
- 2. Métodos com Implementação Padrão:** Ao contrário de Java (antes do Java 8), Kotlin permite que interfaces tenham métodos com implementações padrão. Isso significa que você pode definir um comportamento padrão para um método em uma interface, e as classes implementadoras podem ou não sobrescrever esse método.
- 3. Propriedades:** As interfaces em Kotlin podem conter declarações de propriedades. No entanto, as interfaces não podem armazenar o estado de uma propriedade; elas podem apenas declarar propriedades abstratas ou fornecer acessadores (getters/setters) com implementações padrão.

Interfaces

- ⊕ Quando dizemos que interfaces em **Kotlin** (e em muitas outras linguagens de programação orientadas a objetos) **não** podem armazenar o **estado** de uma propriedade, estamos nos referindo ao fato de que uma interface **não** pode manter valores de estado para as propriedades que declara.



Métodos Abstratos

- ⊕ Métodos abstratos em **Kotlin** são uma parte fundamental do polimorfismo em programação orientada a objetos;
 - ⊕ São declarados em uma **classe abstrata** ou em uma **interface**.
-
1. **Sem Implementação Própria:** Um método abstrato não tem uma implementação própria. Em vez disso, ele fornece apenas a declaração (assinatura do método).
 2. **Obrigação de Implementação:** Classes que herdam uma classe abstrata ou que implementam uma interface com métodos abstratos devem fornecer uma implementação concreta para esses métodos.
 3. **Palavra-chave `abstract`:** Métodos abstratos são declarados com a palavra-chave `abstract`.

Classes Abstratas

- ⊕ Uma **classe abstrata** em **Kotlin** é um conceito chave na programação orientada a objetos;
 - ⊕ Representa uma classe que **não** pode ser instanciada diretamente e é tipicamente usada como uma classe base para outras classes.
1. **Não Instanciável:** Você não pode criar instâncias de uma classe abstrata diretamente.
Seu propósito é servir como uma classe base para outras classes.
 2. **Palavra-chave `abstract`:** Para declarar uma classe abstrata em Kotlin, você usa a palavra-chave **`abstract`**.

```
abstract class MinhaClasseAbstrata {  
    //...  
}
```

Métodos abstratos e concretos

- Uma **classe abstrata** pode conter **métodos abstratos** (sem corpo) e métodos concretos (com corpo). Os métodos abstratos devem ser **sobrescritos** pelas subclasses, enquanto os métodos concretos podem, mas não precisam ser.

```
abstract class Animal {  
    abstract fun fazerSom()  
  
    fun dormir() {  
        println("Dormindo")  
    }  
}
```

Métodos abstratos e concretos

- **Subclasses:** Para usar uma classe abstrata, você cria uma subclasse que herda da classe abstrata e fornece implementações para os métodos e propriedades abstratas

```
class Cachorro : Animal() {  
    override fun fazerSom() {  
        println("Au au")  
    }  
}
```

- **Polimorfismo:** As classes abstratas são frequentemente usadas em situações onde várias classes têm funcionalidades comuns, mas cada uma delas implementa essas funcionalidades de maneira diferente. Isso permite polimorfismo, onde você pode usar uma referência de uma classe abstrata para manter objetos de suas subclasses.

Implementando interfaces

```
interface MyInterface {  
    fun myFunction() // Método abstrato  
  
    fun myDefaultFunction() { // Método com implementação padrão  
        println("Implementação padrão")  
    }  
}  
  
class MyClass : MyInterface {  
    override fun myFunction() {  
        println("Minha implementação de myFunction")  
    }  
    // myDefaultFunction() não precisa ser sobrescrito  
}  
  
fun main() {  
    val myObj = MyClass()  
    myObj.myFunction() // Saída: Minha implementação de myFunction  
    myObj.myDefaultFunction() // Saída: Implementação padrão  
}
```

Interfaces

- Uma classe pode implementar várias interfaces.
- Se uma classe implementa várias interfaces que têm métodos com o mesmo nome e assinatura, a classe precisa fornecer sua própria implementação para resolver a ambiguidade.
- As interfaces não podem ser instanciadas diretamente. Em vez disso, você cria uma instância de uma classe que implementa a interface.

Interfaces em Kotlin são uma ferramenta poderosa para design de software, permitindo a definição de contratos padronizados enquanto oferece flexibilidade na implementação desses contratos pelas classes.

Herança Múltipla

- ⊕ **Kotlin não** suporta herança múltipla de classes, o que significa que uma classe em **Kotlin não** pode herdar diretamente de mais de uma classe;
- ⊕ Essa é uma característica comum em muitas linguagens de programação orientadas a objetos, como **Java**, para evitar certos problemas complexos, como o "**problema do diamante**", que ocorre em heranças múltiplas.

Alternativas para Herança Múltipla

- Kotlin permite que interfaces contenham implementações padrão de métodos.
- Uma classe em Kotlin pode implementar múltiplas interfaces e, portanto, herdar comportamentos de várias fontes.
- Isso permite que uma classe tenha múltiplos "tipos" e implemente métodos de várias interfaces.

Alternativas para Herança Múltipla

```
interface A {  
    fun fazerAlgo() {  
        println("Fazendo algo em A")  
    }  
}  
  
interface B {  
    fun fazerOutraCoisa() {  
        println("Fazendo outra coisa em B")  
    }  
}  
  
class C : A, B {  
    // C pode usar métodos tanto de A quanto de B  
}
```

Tarefa 5.08



Qual o valor total da Folha de Pagamento?

```

open class Colaborador(val nome: String, val horasTrabalhadas: Int, val valorPorHora: Double) {

    open fun calcularSalario(): Double {
        return horasTrabalhadas * valorPorHora
    }
}

class Gerente(nome: String, horasTrabalhadas: Int, valorPorHora: Double, val bonus: Double) : Colaborador(nome, horasTrabalhadas, valorPorHora) {

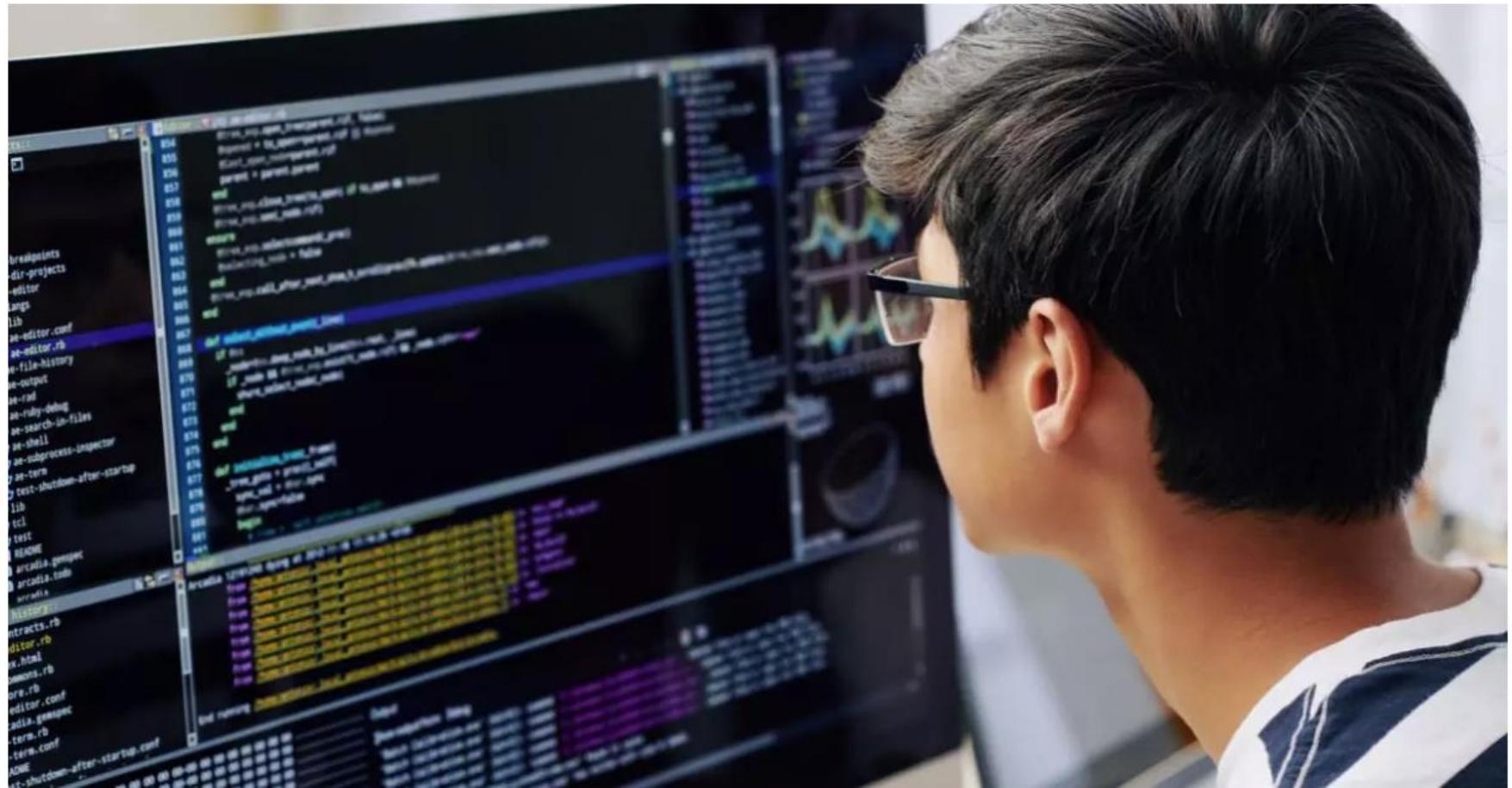
    override fun calcularSalario(): Double {
        return super.calcularSalario() + bonus
    }
}

fun main() {
    val funcionarios: Array<Colaborador> = arrayOf(
        Colaborador("Carlos", 160, 25.0),
        Gerente("Ana", 180, 30.0, 1000.0),
        Colaborador("João", 170, 22.0),
        Gerente("Maria", 150, 35.0, 1500.0)
    )
    var totalSalarios = 0.0
    for (funcionario in funcionarios) {
        totalSalarios += funcionario.calcularSalario()
    }
    println("\nTotal da folha de pagamento: R$$totalSalarios")
}

```

- A) 22760.0
- B) 20857.0
- C) 21456.0
- D) 23560.0
- E) 20890.0

Tarefa 5.09



Qual o valor total dos Produtos ?

```
open class Mercadoria(val nome: String, val preco: Double, val quantidade: Int) {  
    open fun calcularValorTotal(): Double = preco * quantidade  
}  
  
class ProdutoEspecial(nome: String, preco: Double, quantidade: Int, val desconto: Double) :  
    Mercadoria(nome, preco, quantidade) {  
  
    override fun calcularValorTotal(): Double {  
        val valorSemDesconto = super.calcularValorTotal()  
        return valorSemDesconto - (valorSemDesconto * desconto / 100)  
    }  
}  
  
fun main() {  
    val produtos: Array<Mercadoria> = arrayOf(  
        Mercadoria("Notebook", 3500.0, 2),  
        ProdutoEspecial("Smartphone", 2000.0, 3, 10.0),  
        Mercadoria("Teclado", 150.0, 5),  
        ProdutoEspecial("Monitor", 1200.0, 4, 15.0)  
    )  
    var totalValor = 0.0  
    for (produto in produtos) {  
        totalValor += produto.calcularValorTotal()  
    }  
    println("\nTotal do valor dos produtos: R$$totalValor")  
}
```

- A) 17230.0
- B) 18450.0
- C) 16893.0
- D) 17810.0
- E) 18855.0

Tarefa 5.10



Qual o valor total dos produtos com imposto?

```

interface Item {
    val imposto: Double
    fun calcularPrecoFinal(): Double
}

class ItemSimples(val nome: String, val preco: Double) : Item {
    override val imposto: Double = 0.1

    override fun calcularPrecoFinal(): Double {
        return preco + (preco * imposto)
    }
}

class ItemEspecial(val nome: String, val preco: Double, val desconto: Double) : Item {
    override val imposto: Double = 0.15

    override fun calcularPrecoFinal(): Double {
        val precoComDesconto = preco - (preco * desconto / 100)
        return precoComDesconto + (precoComDesconto * imposto)
    }
}

fun main() {
    val itens: Array<Item> = arrayOf(
        ItemSimples("Cadeira", 300.0),
        ItemSimples("Mesa", 1000.0),
        ItemSimples("Ventilador", 200.0),
        ItemEspecial("Computador", 5000.0, 5.0)
    )
    var valorTotal = 0.0
    for (item in itens) {
        valorTotal += item.calcularPrecoFinal()
    }
    println("\nValor total dos produtos com imposto: R$$valorTotal")
}
  
```

- A) 7654.5
- B) 7332.0
- C) 7112.5
- D) 7654.3
- E) 7611.0