Desenvolvimento de Aplicativos Móveis

Professor Maurício Buess
mbuess@up.edu.br
https://github.com/mauriciobuess

Classes e Objetos:

- No **Java**, uma classe é declarada com a palavra-chave **class**, e as propriedades devem ser explicitamente declaradas dentro do corpo da classe.

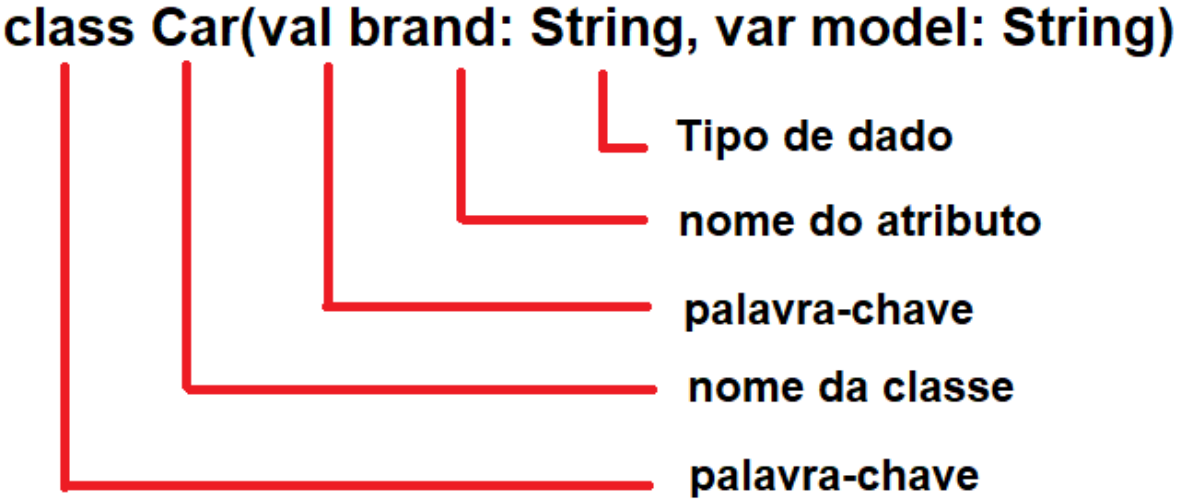
```
public class Car {
    private String brand;
    private String model;
    // Construtor
    public Car(String brand, String model) {
        this.brand = brand;
        this.model = model;
    }
    // Getters e Setters
    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }

    public String getModel() {
        return model;
    }

    public void setModel(String model) {
        this.model = model;
    }
}
```

Classes e Objetos:

- Ainda no Java, a criação de objeto implica o uso da palavra reservada **new**, seguindo uma sintaxe específica da linguagem:

Classes e Objetos:

- No Kotlin, a declaração de classes é muito mais simples e resumida, podendo as propriedades serem declaradas diretamente no construtor primário.

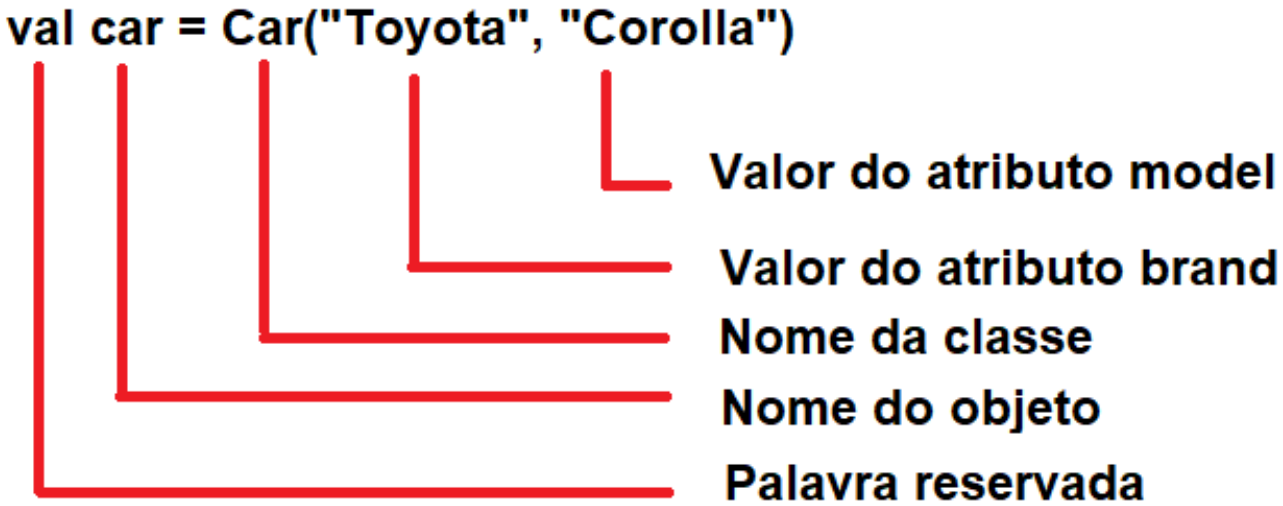


val indica que o atributo **brand** é somente leitura (imutável).

var indica que o atributo m**odel** é somente leitura e gravação (mutável).

Classes e Objetos:

- No Kotlin, não é necessário usar a palavra-chave new para criação (instanciação) do objeto



Principais diferenças:

- Java

- definição de um construtor é feita explicitamente dentro da classe.

- As propriedades de uma classe em Java são declaradas como campos (variáveis de instância) e geralmente são privadas, com getters e setters para acesso.

- o acesso a variáveis e métodos é controlado por private, protected, e public.

- **Kotlin**

- permite a definição de um construtor primário diretamente na assinatura da classe e construtores secundários, se necessário.

class Car(val brand: String, var model: String)

- as propriedades podem ser públicas por padrão e o próprio compilador gera os getters e setters.

- Também tem private, protected, e public, mas adiciona o modificador internal, que torna a classe ou membro visível dentro do mesmo módulo.

Semelhanças:

- Herança
  - Ambos suportam herança, mas em Kotlin, as classes são **final** por padrão (não podem ser herdadas a menos que sejam explicitamente marcadas como **open**).

- Polimorfismo
  - O polimorfismo funciona de maneira similar em ambas as linguagens, permitindo que uma superclasse seja referenciada para instanciar subclasses.

Herança Java:

```
public class Vehicle {
    private String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    public void startEngine() {
        System.out.println("Engine started");
    }
}
```

```
public class Car extends Vehicle {
    private String model;

    public Car(String brand, String model) {
        super(brand);
        this.model = model;
    }

    @Override
    public void startEngine() {
        System.out.println("Car engine started");
    }
}
```

Herança Kotlin:

```
open class Vehicle(val brand: String) {
    open fun startEngine() {
        println("Engine started")
    }
}

class Car(brand: String, val model: String) : Vehicle(brand) {
    override fun startEngine() {
        println("Car engine started")
    }
}
```

Data class (Kotlin):

- Classe especial cuja principal finalidade é armazenar dados.
- O compilador Kotlin fornece automaticamente várias funcionalidades para as data classes, como geração de equals(), hashCode(), toString(), e métodos de cópia (copy()), bem como suporte para destructuring.

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

- **equals()**: Compara dois objetos para verificar se possuem os mesmos valores em suas propriedades.
- **hashCode()**: Retorna um código hash baseado nos valores das propriedades.
- **toString()**: Gera uma string representando a classe, por exemplo: "Person(name=John, age=30)".
- **copy()**: Cria uma nova instância da classe com os mesmos valores, permitindo alterar propriedades específicas.
- **Destructuring**: Permite decompor a instância em variáveis individuais.

Data class (Kotlin):

```
val person1 = Person("Joao", 30)
val person2 = person1.copy(age = 31)

// Destructuring
val (name, age) = person1

// imprime: Joao tem 30 anos
println("$name is $age years old")
```

- Criação do objeto: person1 é criado como uma instância de Person com o nome "John" e a idade 30.
- Cópia e modificação: person2 é uma nova instância de Person baseada em person1, mas com a idade modificada para 31.
- Destructuring: O objeto person1 é desestruturado para obter name e age, facilitando a manipulação dessas propriedades individualmente.
- Impressão: O nome e a idade são impressos no formato especificado.

Data class com Java

```
public class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Person person = (Person) o;

        if (age != person.age) return false;
        return name.equals(person.name);
    }
    @Override
    public int hashCode() {
        int result = name.hashCode();
        result = 31 * result + age;
        return result;
    }
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

    public Person copy(String name, int age) {
        return new Person(name != null ? name : this.name, age != 0 ? age : this.age);
    }
}
```

Data class com Java

- Implementar manualmente os métodos equals(), hashCode(), e toString().
- Criar um construtor.
- Escrever getters para as propriedades.
- Implementar um método copy() para criar uma nova instância com possíveis modificações.

Data class Kotlin

- data classes são uma das funcionalidades mais poderosas e eficientes de Kotlin, facilitando a criação de classes voltadas ao armazenamento de dados, reduzindo a quantidade de código boilerplate que seria necessário em Java.
- Esse recurso também promove código mais legível e menos propenso a erros.

Exercício 1: Criação de Classes

Você foi contratado para criar um sistema simples de gerenciamento de uma biblioteca. A primeira tarefa é criar uma classe Livro que represente um livro. Cada livro deve ter um título, um autor e um número de páginas. Em seguida, crie uma instância dessa classe e exiba as informações do livro no console.

Dicas:

- Defina uma classe Livro em Kotlin com propriedades para título, autor, e número de páginas.
- Use um construtor primário para inicializar essas propriedades.
- Instancie um objeto Livro e utilize a função println() para exibir as informações do livro.

Exercício 1: Criação de Classes

```
class Livro(val titulo: String
        , val autor: String
        , val paginas: Int)

fun main() {
    val livro = Livro("Programando em Kotlin", "Rolando Caio da Rocha", 320)

    println("Título: ${livro.titulo}, Autor: ${livro.autor}, Pages: ${livro.paginas}")
}
```

Exercício 2: Herança

Em um sistema de gerenciamento de veículos, você precisa criar uma hierarquia de classes. Comece criando uma classe base chamada Veiculo com propriedades como marca e ano. Depois, crie uma classe derivada chamada Carro que, além das propriedades de Veiculo, tenha uma propriedade específica chamada numberDePortas. Crie instâncias dessas classes e exiba as informações.

Dicas:

- Crie a classe Veiculo com as propriedades marca e ano.
- Use a palavra-chave open para permitir que a classe Veiculo seja herdada.
- Crie a classe Carro que herda de Veiculo e adicione a propriedade numberDePortas.
- Instancie objetos de ambas as classes e exiba seus detalhes.

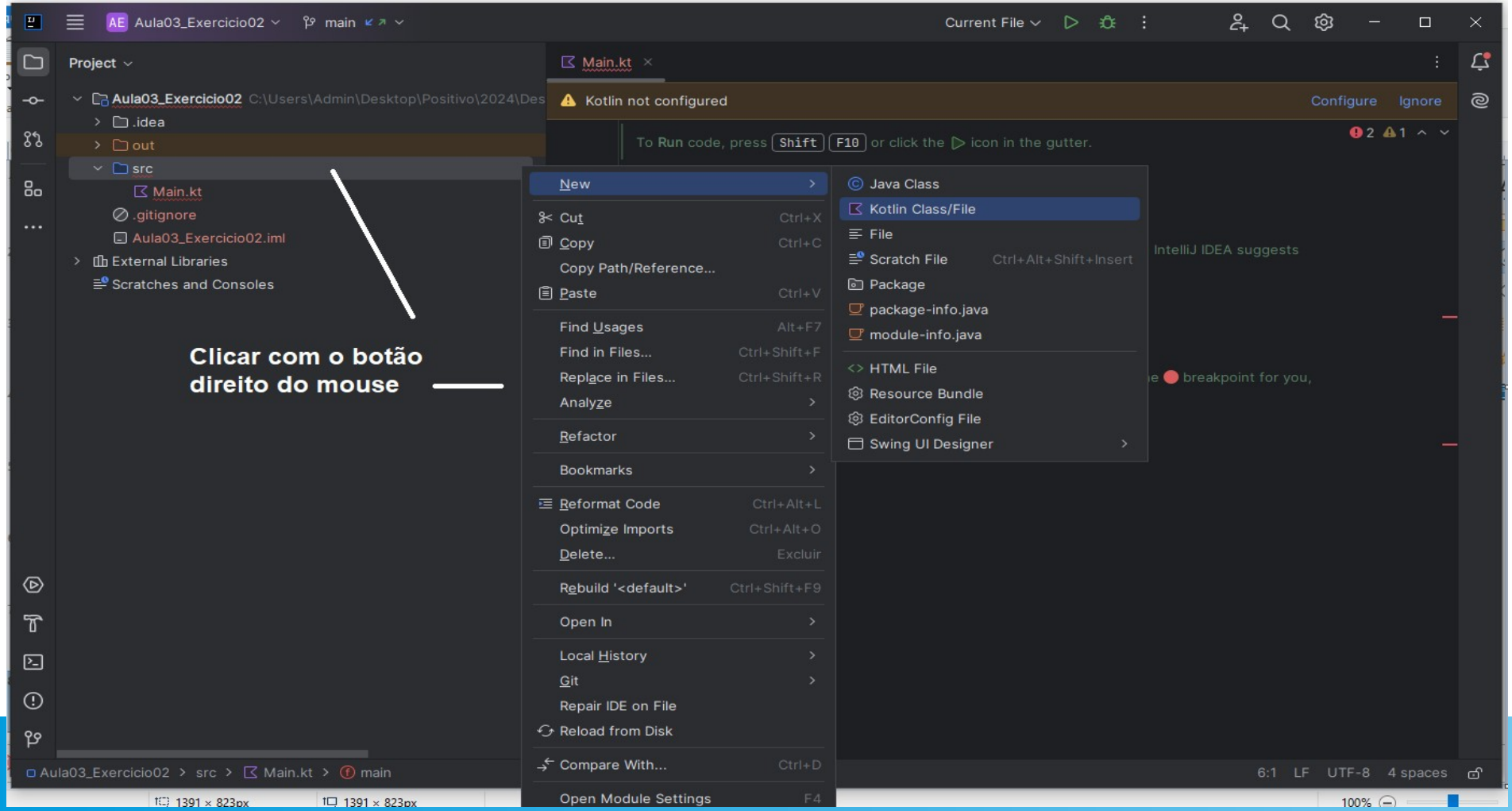
Exercício 2: Herança

Em um sistema de gerenciamento de veículos, você precisa criar uma hierarquia de classes. Comece criando uma classe base chamada Veiculo com propriedades como marca e ano. Depois, crie uma classe derivada chamada Carro que, além das propriedades de Veiculo, tenha uma propriedade específica chamada numberDePortas. Crie instâncias dessas classes e exiba as informações.

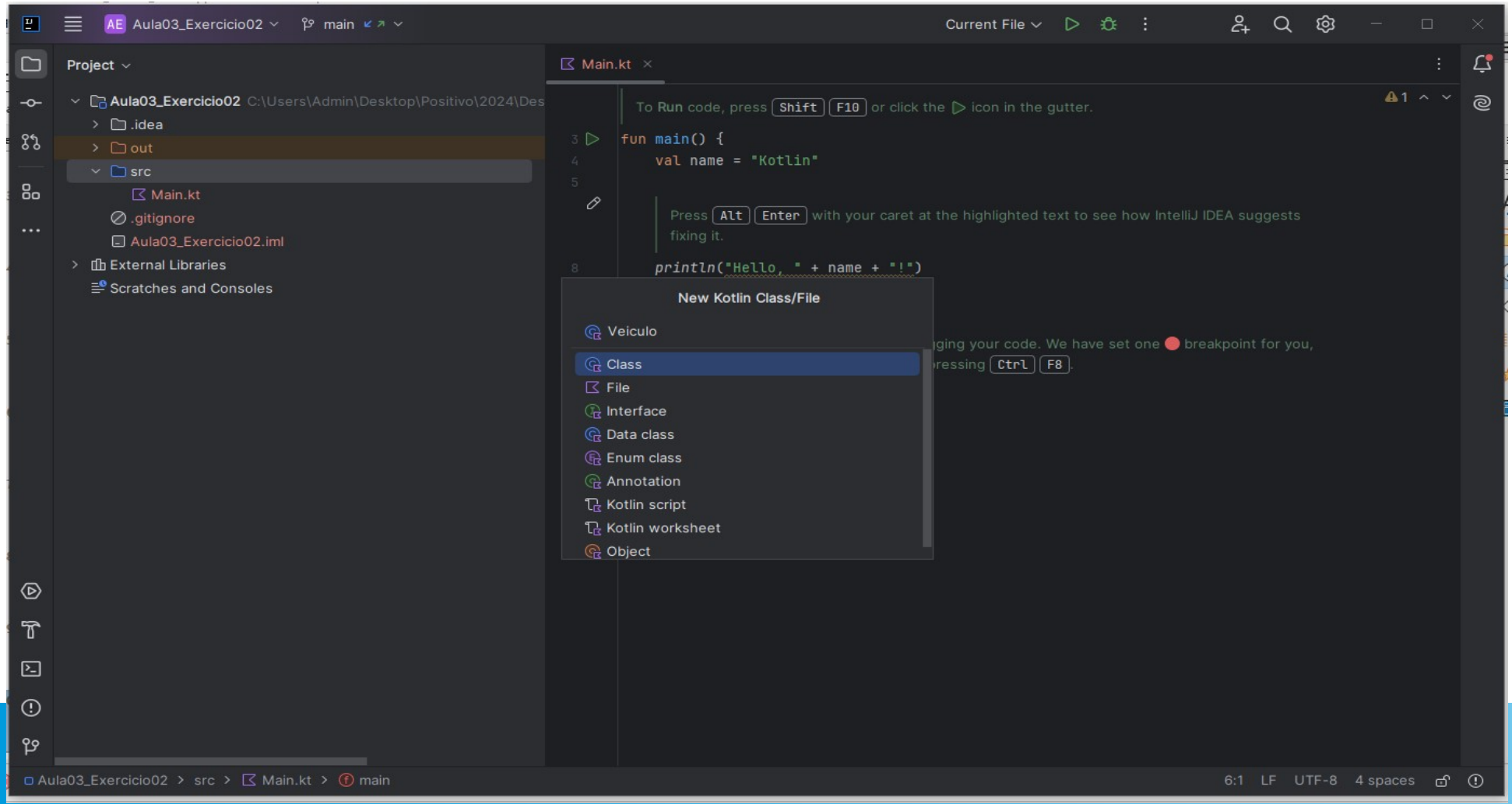
Dicas:

- Crie a classe Veiculo com as propriedades marca e ano.
- Use a palavra-chave open para permitir que a classe Veiculo seja herdada.
- Crie a classe Carro que herda de Veiculo e adicione a propriedade numberDePortas.
- Instancie objetos de ambas as classes e exiba seus detalhes.

Desenvolvimento Mobile



Desenvolvimento Mobile



- Arquivo Veiculo.kt

```
open class Veiculo(val marca: String, val ano: Int)

class Carro(marca: String, ano: Int, val numeroDePortas: Int) : Veiculo(marca, ano)
```

- Arquivo Main.kt

```
fun main() {
    val veiculo = Veiculo("Toyota", 2020)
    val carro = Carro("Honda", 2021, 4)

    println("Veiculo: Marca = ${veiculo.marca}, Ano = ${veiculo.ano}")
    println("Carro: Narca = ${carro.marca}, Ano = ${carro.ano}, Portas = ${carro.numeroDePortas}")
}
```

Exercício 3: Polimorfismo

Uma empresa de entregas possui diferentes tipos de veículos: Bike e Truck. Todos os veículos têm um método deliver() que descreve como a entrega é feita. A ideia é criar uma interface Deliverable com um método deliver(), e fazer com que Bike e Truck implementem essa interface. Implemente as classes e demonstre o polimorfismo através de uma função que recebe uma lista de veículos e chama deliver() para cada um deles.

Dicas:

- Crie uma interface Deliverable com o método deliver().
- Implemente a interface nas classes Bike e Truck.
- Crie uma função que receba uma lista de Deliverable e chame deliver() para cada item.
- Instancie objetos de Bike e Truck e passe-os para a função.

- Arquivo Interfaces.kt

```
interface Deliverable {
    fun deliver()
}
```

- Arquivo Classes.kt

```
class Bike : Deliverable {
    override fun deliver() {
        println("Entrega por bicicleta.")
    }
}

class Truck : Deliverable {
    override fun deliver() {
        println("Entrega por caminhão.")
    }
}
```

- Arquivo Main.kt

```
fun main() {
    val bike = Bike()
    val truck = Truck()

    startDelivery(listOf(bike, truck))
}

fun startDelivery(deliverables: List<Deliverable>) {
    for (deliverable in deliverables) {
        deliverable.deliver()
    }
}
```

Exercício 4: Classes Abstratas

Uma empresa de software está desenvolvendo um sistema de pagamento online. No sistema, existem diferentes métodos de pagamento, como CreditCard e PayPal. Crie uma classe abstrata Payment que tenha um método abstrato processPayment(). Em seguida, crie classes concretas para CreditCard e PayPal que implementem este método.

Dicas:

- Crie uma classe abstrata Payment com o método abstrato processPayment().
- Implemente classes concretas CreditCard e PayPal que herdam de Payment e implementam o método processPayment().
- Instancie objetos de CreditCard e PayPal e chame processPayment() para cada um.

Exercício 4: Classes Abstratas

- **Arquivo: Classes.kt**

```
abstract class Payment {
    abstract fun processPayment(amount: Double)
}

class CreditCard : Payment() {
    override fun processPayment(amount: Double) {
        println("Pgto.cartão de crédito R$ $$amount")
    }
}

class PayPal : Payment() {
    override fun processPayment(amount: Double) {
        println("Pgto.PayPal de $$amount")
    }
}
```

- Arquivo Main.kt

```
fun main() {
    val payment1: Payment = CreditCard()
    val payment2: Payment = PayPal()

    payment1.processPayment(100.0)
    payment2.processPayment(200.0)
}
```

Exercício 5: Utilizando Data Classes

Uma empresa deseja criar um sistema de gerenciamento de clientes onde cada cliente possui um nome, um e-mail e uma idade. A tarefa é criar uma data class em Kotlin chamada Customer e, em seguida, criar uma lista de clientes. Implemente uma função que filtre a lista para exibir apenas os clientes maiores de idade.

Dicas:

- Crie a data class Customer com as propriedades name, email, e age.
- Crie uma lista de objetos Customer.
- Implemente uma função que filtre a lista e exiba apenas os clientes maiores de idade (idade ≥ 18).

Desenvolvimento Mobile

- **Arquivo Classes.kt**

```
data class Customer(val name: String
                  , val email: String
                  , val age: Int)

val customers = listOf(
    Customer("Alice", "alice@example.com", 23),
    Customer("Bob", "bob@example.com", 17),
    Customer("Charlie", "charlie@example.com", 30)
)
```

- **Arquivo Main.kt**

```
fun main() {
    val adultCustomers = customers.filter { it.age >= 18 }
    println("Clientes adultos: $adultCustomers")
}
```

Introdução à Programação Funcional em Kotlin

- Kotlin permite combinar paradigmas de programação orientada a objetos (OOP) com programação funcional (FP). A linguagem oferece recursos como funções *lambda* e funções de *ordem superior*, que facilitam a expressão de comportamentos e a manipulação de funções como objetos de primeira classe.

- **Funções Lambda**

  - Funções lambda são funções anônimas que podem ser tratadas como valores e passadas como parâmetros para outras funções. Elas são úteis para expressar pequenos trechos de código que serão executados como comportamentos específicos.

Exemplo:

```
Fun main() {
    val sum = { a: Int, b: Int -> a + b }
    println(sum(3, 4)) // Output: 7
}
```

- **Uso Prático Lambda:**

- Imagine que você precisa filtrar uma lista de números para encontrar os que são maiores que um determinado valor. Você pode usar uma lambda para expressar esse comportamento:

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 5)
    val filteredNumbers = numbers.filter { it > 3 }
    println(filteredNumbers) // Output: [4, 5]
}
```

- **Funções de Ordem Superior**

- São funções que recebem outras funções como parâmetros ou retornam funções como resultado. Elas permitem uma flexibilidade maior no design do código, promovendo a reutilização e a abstração de comportamentos comuns.

Exemplo:

Suponha que você queira criar uma função que aplique uma transformação a cada item de uma lista. Essa função pode receber uma lambda como parâmetro para definir a transformação.

- **Funções de Ordem Superior**

- Suponha que você queira criar uma função que aplique uma transformação a cada item de uma lista, essa tranformação se rezume em dobrar o valor de cada posição da lista. Essa função pode receber uma lambda como parâmetro para definir a transformação.

```
fun <T, R> transformList(list: List<T>, transform: (T) -> R): List<R> {
    return list.map(transform)
}

fun main() {
        val numbers = listOf(1, 2, 3, 4, 5)
        val doubled = transformList(numbers) { it * 2 }
        println(doubled) // Output: [2, 4, 6, 8, 10]
}
```

```
fun <T, R> transformList(list: List<T>, transform: (T) -> R): List<R> {
    return list.map(transform)
}
```

- **<T, R>**: Parâmetros de tipo genérico. T representa o tipo dos elementos na lista de entrada e R representa o tipo dos elementos na lista de retorno (após operação).
- **list: List<T>**: O primeiro parâmetro é uma lista de elementos do tipo T.
- **transform: (T) -> R**: O segundo parâmetro é uma função lambda que recebe um argumento do tipo T (que é uma lista) e retorna um valor do tipo R. Esta função é aplicada a cada elemento da lista para produzir a lista resultante.
- **return list.map(transform)**: A função map é uma função de extensão disponível para listas em Kotlin. Ela aplica a função transform a cada elemento da lista list e retorna uma nova lista contendo os resultados

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 5)
    val doubled = transformList(numbers) { it * 2 }
    println(doubled) // Output: [2, 4, 6, 8, 10]
}
```

val numbers = listOf(1, 2, 3, 4, 5): Cria uma lista imutável com números 1 a 5

val doubled = transformList(numbers) { it * 2 }: Chama a função transformList passando a lista numbers e uma lambda como argumentos. A lambda { it * 2 } define a função de transformação que multiplica cada número por 2. O resultado é uma nova lista onde cada número da lista original foi dobrado.

it: palavra-chave em Kotlin que representa o parâmetro único de uma lambda quando não é nomeado explicitamente. Nesse caso, it representa cada elemento da lista numbers.

```
fun main() {
    val numbers = listOf(1, 2, 3, 4, 5)

    val stringNumbers = transformList(numbers) { it.toString() }
    println(stringNumbers) // Output: [1, 2, 3, 4, 5]
}
```

- **transformList(numbers) { it.toString() }**: transformList é uma função lambda que converte um Int para uma String. Então, a lista de inteiros [1, 2, 3, 4, 5] é transformada em uma lista de strings ["1", "2", "3", "4", "5"].