



Desenvolvimento Mobile

Orientação a Objetos

Professor MSc. Antônio Catani
antonio.catani@faculdadeimpacta.com.br

Classes e Objetos

Classes e Objetos

- Classe:
 - Especificação para um ou mais objetos.
 - Descrição de um conjunto de atributos e comportamentos.
- Objeto ou Instância:
 - Pertence a uma classe.
 - Realização ou instanciação de uma classe.
 - Instanciação = criação de um objeto.

Classes e Objetos

- Para criar uma classe basta utilizar a palavra reservada `class`.
 - O construtor padrão é definido na mesma linha da classe.
- Não é preciso utilizar `new` para criar uma instância da classe.
- Para saber mais:

<https://kotlinlang.org/docs/reference/classes.html>

Classes e Objetos

- Exemplo:


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

    init {
        // bloco de inicialização
        this.nome = nome
        this.idade = idade
    }

    override fun toString(): String {
        return "Aluno: $nome,
                idade: $idade"
    }
}
```

```
fun main(args: Array<String>) {
    val a1 = Aluno("Carmen", 20)
    println(a1)
    println("Aluno: ${a1.nome},
            idade: ${a1.idade}")
}
```

1.6.21 ▾ JVM ▾ Program arguments

 [Copy link](#)

[↔ Share code](#)

 Run

```
/* Classes */
package org.kotlinlang.play

class Aluno(nome: String, idade: Int) {
    val nome: String
    val idade: Int

    init {
        // bloco de inicialização
        this.nome = nome
        this.idade = idade
    }

    override fun toString(): String {
        return "Aluno: $nome, idade: $idade"
    }
}
```

```
fun main(args: Array<String>) {  
    val a1 = Aluno("Carmen", 20)  
    println(a1)  
    println("Aluno: ${a1.nome}, idade: ${a1.idade}")  
}
```

```
/* Classes */
package org.kotlinlang.play

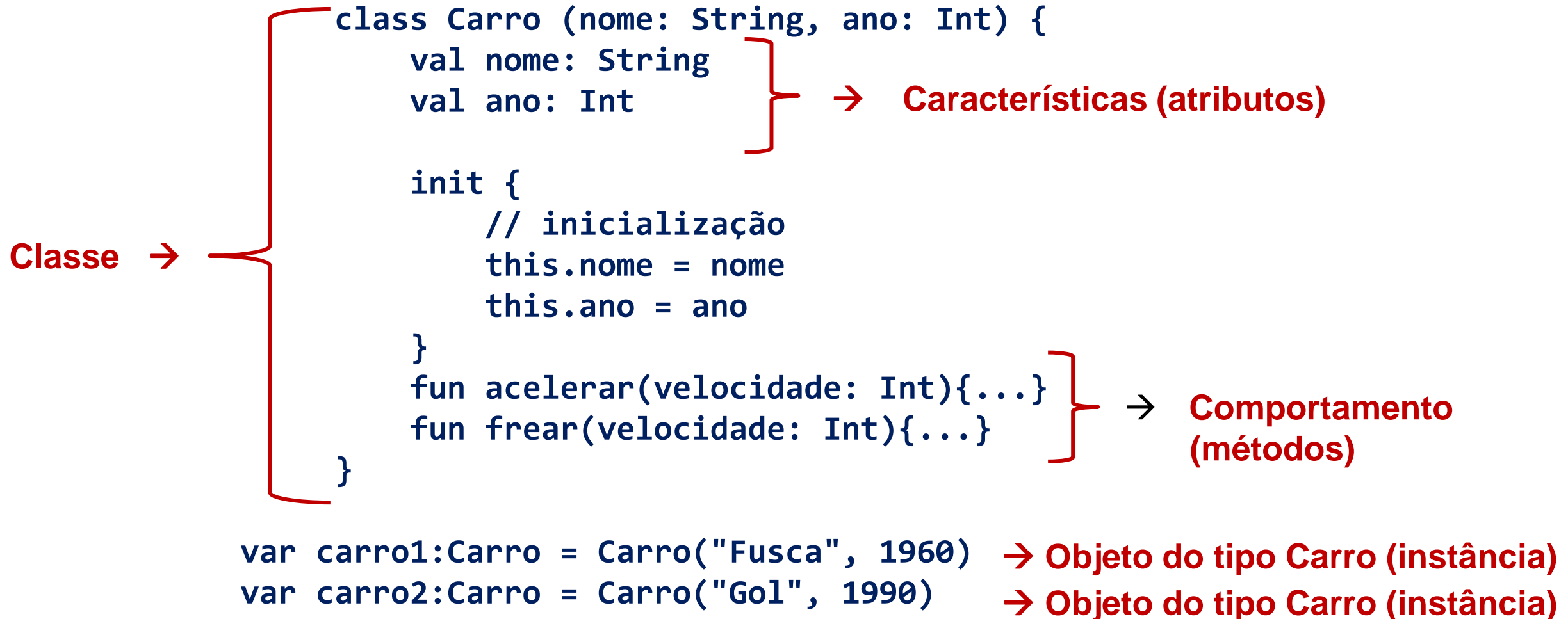
class Aluno(nome: String, idade: Int) {
    val nome: String
    val idade: Int

    init {
        // bloco de inicialização
        this.nome = nome
        this.idade = idade
    }
}
```

Aluno: Carmen, idade: 20
 Aluno: Carmen, idade: 20

Classes e Objetos

- Exemplo em Kotlin: **Classe: carro** - **Objetos: carro1 e carro2.**



Classes e Objetos

- Pergunta: considerando a classe anterior e o código abaixo, responda:

```
var carro1:Carro = Carro('Fusca', 1960)
```

```
var carro2:Carro = Carro('Fusca', 1960)
```

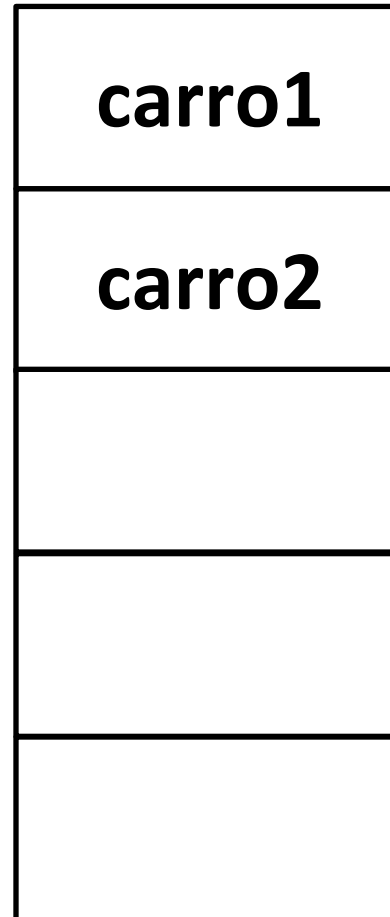
- carro1 e carro2 são o mesmo carro?

Classes e Objetos

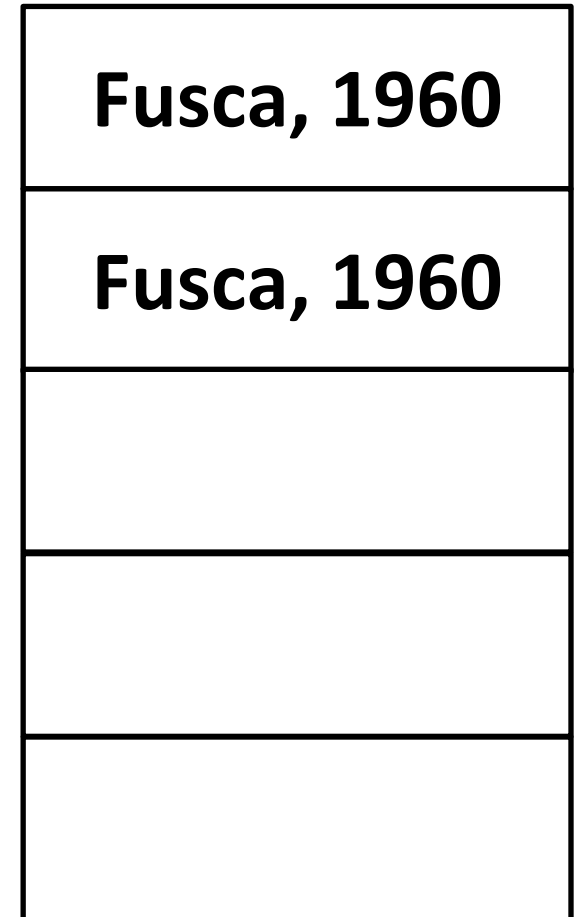
```
var carro1:Carro = Carro("Fusca", 1960)
```

```
var carro2:Carro = Carro("Fusca", 1960)
```

Memória Estática
(Stack)



Memória Dinâmica
(Heap)



Herança

Herança

- Herança permite que um código definido em uma classe (superclasse) seja reutilizado por outra (subclasse).
- Superclasses não sabem nada sobre as subclasses que herdam dela.
- Uma subclasse conhece tudo que está acessível da superclasse (variáveis de instâncias e métodos).
- Dizemos que a subclasse herda da superclasse.
- Uma subclasse pode substituir os métodos da superclasse para especificar os comportamentos (*overriding*).
- Herança define um relacionamento é-um (*is-a*).

Herança

- Para utilizar herança basta utilizar `:` e o nome da classe mãe (superclass).
- Para que uma classe possa ser herdada ela deve ser marcada como open.
 - Todas as classes são final por padrão.
 - A mesma regra vale para métodos.
 - O método `toString` foi sobrescrito para ser chamado sempre que a classe for convertida em `String`.
- Para saber mais:

<https://kotlinlang.org/docs/reference/classes.html>

Herança

- Exemplo:

```
open class Pessoa(nome: String, idade: Int) {
    val nome: String val idade: Int
    init {
        // bloco de inicialização
        this.nome = nome
        this.idade = idade
    }
    open fun adicionarDesconto(desconto: Int) {
        println("Desconto para a pessoa de $desconto")
    }
    override fun toString(): String {
        return "Pessoa: $nome, idade: $idade"
    }
}
```

Herança

```
class Aluno(nome: String, idade: Int): Pessoa(nome, idade) {  
    override fun adicionarDesconto(desconto: Int) {  
        println("Desconto para o aluno de $desconto")  
    }  
}  
  
fun main(args: Array<String>) {  
    val a1 = Aluno("Carmen", 20)  
    println(a1)  
    println("Aluno: ${a1.nome}, idade: ${a1.idade}")  
    a1.adicionarDesconto(10)  
}
```



```
/* Herança */
package org.kotlinlang.play

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

    init {
        // bloco de inicialização
        this.nome = nome
        this.idade = idade
    }

    open fun adicionarDesconto(desconto: Int) {
        println("Desconto para a pessoa de $desconto")
    }
}
```

1.6.21 ▾

JVM ▾

Program arguments

🔗 Copy link

<> Share code

▶ Run

```
|
override fun toString(): String {
    return "Pessoa: $nome, idade: $idade"
}

class Aluno(nome: String, idade: Int): Pessoa(nome, idade) {
    override fun adicionarDesconto(desconto: Int) {
        println("Desconto para o aluno de $desconto")
    }
}

fun main(args: Array<String>) {
    val a1 = Aluno("Carmen", 20)
    println(a1)
    println("Aluno: ${a1.nome}, idade: ${a1.idade}")
    a1.adicionarDesconto(10)
}
```

?

```
/* Herança */
package org.kotlinlang.play

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

    init {
        // bloco de inicialização
        this.nome = nome
        this.idade = idade
    }
}
```

Pessoa: Carmen, idade: 20
 Aluno: Carmen, idade: 20
 Desconto para o aluno de 10

Herança

- Exemplo Superclasse Carro e subclasse Ferrari (Ferrari é um Carro).

```
open class Carro (nome: String, ano: Int) {

    val nome: String
    val ano: Int

    init {
        // inicialização
        this.nome = nome
        this.ano = ano
    }

    open fun acelerar(velocidade: Int) {...}
    open fun frear(velocidade: Int) {...}
}

class Ferrari (nome: String,
               ano: Int): Carro(nome, ano) {
    // método acelerar é sobrescrito
    override fun acelerar(velocidade: Int) {...}
    // método frear é herdado da superclasse
}
```

Herança

- Em Kotlin, três coisas são importantes ao trabalhar com herança:
 - A classe deve ter o modificador open, uma vez que por padrão todas as classes são final.
 - Os métodos que podem ser herdados também devem ser marcados com open.
 - Para sobrescrever um método herdado, este método deve ser marcado com override.

Interfaces

Interfaces

- Caso especial de uma classe abstrata.
- Não precisa (mas pode) implementar os métodos, apenas define seu nome e parâmetros.
- Uma interface define propriedades e métodos que uma subclasse deve implementar.
- Uma interface é como um contrato:
 - A subclasse não abstrata que implementa a interface deve prover a implementação dos métodos abstratos declarados na interface.

Interfaces

- Exemplo: Classe Carro e Interface Aceleravel.

```
interface Aceleravel {  
    // método abstrato, sem implementação  
    fun acelerar()  
}
```

```
class Carro: Aceleravel {  
    // carro deve prover a implementação de acelerar  
    override fun acelerar() {...}  
}
```

- Implementar uma interface funciona como uma herança.
 - Portanto, Carro é-um Aceleravel.

Instanciação

Instanciação

- Instanciação:
 - Instanciar é criar um objeto de uma classe.
 - A instanciação de objetos em Kotlin difere o tipo da referência (variável) e o tipo da instância (objeto).
 - `var carro1:Carro = Carro('Fusca', 1960);`

```
var carro1:Carro = Carro("Fusca", 1960)
```



Tipo da referência



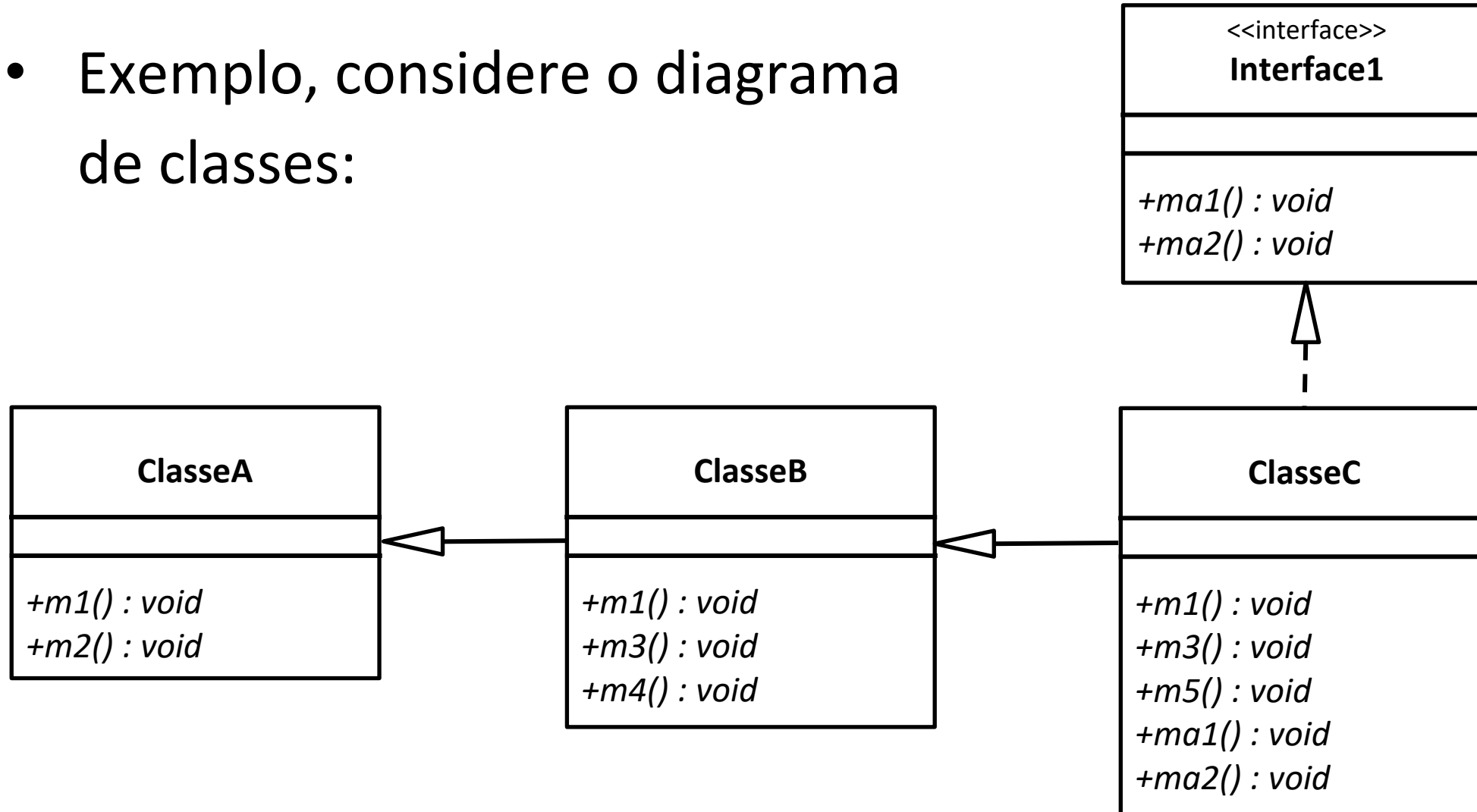
Tipo da instância

Instanciação

- Isso permite que o tipo da referência (variável) seja diferente do tipo da instância (objeto), desde que o tipo da instância seja subclasse ou implemente o tipo da referência.
- Também permite que uma função declare como parâmetro um tipo de uma interface ou de uma superclasse (tipo da referência – parâmetro da função) seja diferente do tipo da instância (objeto) enviado, desde que o tipo da instância seja subclasse ou implemente o tipo da referência.

Instanciação

- Exemplo, considere o diagrama de classes:



Instanciação

- ClasseC implementa Interface1.
- ClasseB herda ClasseA.
 - ClasseB é uma ClasseA.
- ClasseC herda ClasseB.
 - ClasseC é uma ClasseB e uma Classe A.

Instanciação

- Nesse diagrama, as seguintes instanciações são válidas:

```
var objeto:Interface1 = ClasseC()
```

```
var objeto:ClasseC = ClasseC()
```

```
var objeto:ClasseB = ClasseC()
```

```
var objeto:ClasseA = ClasseC()
```

- ClasseC é subclasse de ClasseA e ClasseB, e implementa Interface1.
 - O tipo da referência é uma superclasse ou interface, e o tipo da instância é a subclasse.

Instanciação

- Entretanto, as seguintes instanciações não são válidas:

`var objeto:ClasseC = Interface1()` Não é possível instanciar uma Interface

`var objeto:ClasseC = ClasseB()`

`var objeto:ClasseC = ClasseA()`

Tipo da instância é a superclasse,
enquanto o tipo da referência é a subclasse

`var objeto:ClasseB = ClasseA()`

- Lembre-se: a superclasse não sabe nada sobre a subclasse, então é possível atribuir uma instância da superclasse para um tipo da subclasse (ClasseA não é uma ClasseC, por exemplo).

Orientação a Objetos

- Também é possível criar uma instância da ClasseC e enviá-la para um método que receba um parâmetro dos tipos Interface1, ClasseC, ClasseB e ClasseA:

```
var objeto = ClasseC()
```

```
fun metodo1(arg: Interface1) {...}
```

```
fun metodo2(arg: ClasseC) {...}
```

```
fun metodo3(arg: ClasseB) {...}
```

```
fun metodo1(arg: ClasseA) {...}
```

```
// todas estas chamadas são válidas metodo1(objeto)
```

```
// objeto é uma InterfaceI metodo2(objeto)
```

```
// objeto é uma ClasseC metodo3(objeto)
```

```
// objeto é uma ClasseB metodo4(objeto)
```

```
// objeto é uma ClasseA
```


Instanciação

- ClasseC é subclasse de ClasseA e ClasseB, e implementa InterfaceI.
 - O tipo declarado nos métodos (referência) é a própria ClasseC, uma superclasse ou interface.
 - O tipo enviado (tipo da instância) é a subclasse ou a própria classe.

Instanciação

- Entretanto, as seguintes linhas de comando não são válidas:

```
val objeto = Interface1()
```

Não é possível instanciar uma Interface

```
fun metodo1(arg: ClasseC)
```

```
fun metodo2(arg: ClasseB)
```

```
val objeto:= ClasseB()
```

```
metodo1(objeto)
```

**Tipo da instância é a superclasse,
enquanto o tipo do parâmetro
do método é a subclasse**

```
val objeto = ClasseA()
```

```
metodo1(objeto)
```

```
metodo2(objeto)
```

- Lembre-se: a superclasse não sabe nada sobre a subclasse, então é possível atribuir uma instância da superclasse para um tipo da subclasse (ClasseA não é uma ClasseC, por exemplo).

Data Classes

Data Classes

- Data Classes são classes que contém somente informações.
- Em Kotlin uma Data Class é criada apenas com uma linha.

```
data class Aluno(val nome: String)
```

- Essa linha cria a classe Aluno com:
 - Atributos do construtor.
 - Getters e setters.
 - Métodos equals, toString e copy.
- Para sabe mais:

<https://kotlinlang.org/docs/reference/dataclasses.html>

Orientação a Objetos

- Porque tudo isso?
 - A biblioteca do Android contém muita coisa pronta, sendo que muitas vezes será necessário estender um classe ou implementar uma Interface.
 - A principal classe de um projeto, Activity, é baseada em herança e sobrescrita de métodos.



Obrigado!

Prof. MSc. Antônio Catani
antonio.catani@faculdadeimpacta.com.br