Cábula de Java POO e UML · Programação orientada a objetos

Classes · Construtor · Herança · Interface · Listas

Holger Sebastião · www.sebastiao.org
Versão 1.0 · 2026-10-25

Classe = planta

```java
// Ficheiro: Animal.java
public class Animal {
  // atributos: private
  private String nome;
  private int idade;

  // construtor
  public Animal(String nome, int idade) {
    this.nome = nome;
    this.idade = idade;
  }

  // métodos: public
  public String getNome() {
    return nome;
  }

  public void fazerSom() {
    System.out.println("...");
  }
}
```

Classe	planta, por exemplo `Animal`
Objeto	um exemplar feito segundo a planta (instância), por exemplo `bobi`
Atributo	propriedade: o que um objeto tem
Método	capacidade: o que um objeto sabe fazer

Nas nossas aulas: atributos `private`, métodos `public`. Uma classe por ficheiro, nome do ficheiro = nome da classe.

Criar objetos `new`

```java
Animal bobi = new Animal("Bobi", 3);
Animal mia = new Animal("Mia", 5);

bobi.fazerSom();            // chamar um método
String n = bobi.getNome();  // "Bobi"
```

`new` cria um objeto novo, chama o construtor e devolve uma referência para ele. A variável guarda só essa referência.

```java
Animal a = bobi;    // nenhum objeto novo!
                    // a e bobi apontam
                    // para o mesmo animal
```

Uma variável sem objeto é `null`. `a.fazerSom()` → `NullPointerException`.

`==` verifica se é o **mesmo** objeto. `equals` só compara conteúdos se a classe o reescrever (a String fá-lo).

Construtor

Tem o nome da classe, **não tem tipo de retorno** (nem `void`), corre com `new` e define os valores iniciais.

CONSTRUTOR POR DEFEITO

Sem construtor próprio, o Java cria um vazio: `public Animal() { }`. Os atributos ficam `0`, `false` ou `null`.

Se houver um construtor próprio, o construtor por defeito desaparece. `new Animal()` só funciona se o escreveres tu (em baixo).

CONSTRUTORES PRÓPRIOS

```java
public Animal(String nome, int idade) {
  this.nome = nome;   // atributo = parâmetro
  this.idade = idade;
}

public Animal() {        // sem parâmetros
  this.nome = "Desconhecido";
  this.idade = 0;
}
```

`this.nome` é o atributo do objeto atual, `nome` o parâmetro. Vários construtores = sobrecarga.

Getters e setters `encapsulamento`

```java
public int getIdade() { return idade; }
public void setIdade(int idade) {
  if (idade >= 0) {          // validar
    this.idade = idade;
  }
}
```

Encapsulamento: atributos `private`, acesso só através de métodos. O setter recusa valores errados. Em boolean: `isPronto()`.

Classe abstrata `abstract`

```java
public abstract class Forma {
  // método abstrato: só o cabeçalho
  public abstract double area();
}
// Circulo extends Forma tem de escrever area()
```

Não tem objetos: `new Forma()` não funciona, é só superclasse. As subclasses **têm de** reescrever os métodos abstratos. UML: nome em *itálico*.

Ao contrário de uma interface, pode ter atributos, construtores e métodos prontos. Mas só se herda de **uma** classe.

Herança `extends`

```java
public class Cao extends Animal {
  private String raca;

  public Cao(String nome, int idade,
             String raca) {
    super(nome, idade);  // primeiro: o Animal
    this.raca = raca;
  }

  @Override
  public void fazerSom() {
    System.out.println("Au-au");
  }
}
```

Um cão **é um** animal: herda todos os atributos e métodos e pode ter mais. `Animal` é a **superclasse**, `Cao` a **subclasse**.

`super(…)` chama o construtor da superclasse e tem de ser a primeira linha. `super.fazerSom()` chama o método antigo.

Só é permitida **uma** superclasse. Os atributos privados da superclasse só se alcançam em Cao através de getters.

Todas as classes herdam de `Object`, por isso `toString()` e `equals()` existem em todo o lado.

Sobrecarga e reescrita

SOBRECARGA (OVERLOADING)

Mesmo nome, **outra lista de parâmetros**, normalmente na mesma classe. Só outro tipo de retorno não chega.

```java
public void alimentar() { … }
public void alimentar(int quantidade) { … }
```

REESCRITA (OVERRIDING)

A subclasse volta a escrever um método herdado com o **mesmo cabeçalho**. `@Override` faz o compilador verificar se ele existe mesmo em cima.

```java
@Override
public String toString() {
  return nome + " (" + idade + ")";
}
// println(bobi) → Bobi (3)
```

POLIMORFISMO

```java
Animal a = new Cao("Rex", 2, "Teckel");
a.fazerSom();  // Au-au: método de Cao
```

Interface `implements`

```java
public interface Estimacao {
  void acariciar();  // só cabeçalho, sem corpo
}

public class Cao extends Animal
                 implements Estimacao {
  @Override
  public void acariciar() {
    System.out.println("abana a cauda");
  }
}
```

Uma interface é um **contrato**: define *que* métodos uma classe tem de ter, mas não *como*. Os métodos são automaticamente `public abstract`. Uma classe pode implementar **várias** interfaces: `implements A, B`.

`new Estimacao()` não funciona. Mas: `Estimacao e = new Cao(…);`

ArrayList `lista que cresce`

```java
import java.util.ArrayList;
ArrayList<String> nomes = new ArrayList<>();
nomes.add("Ada");          // juntar no fim
nomes.add(0, "Max");       // na posição 0
String n = nomes.get(0);   // "Max"
nomes.set(1, "Eva");       // substituir pos. 1
nomes.remove(0);           // apagar pos. 0
int total = nomes.size();
nomes.contains("Eva")      // true
nomes.isEmpty()            // false: não vazia
nomes.clear();             // apagar tudo
```

	array: `a.length`	lista: `lista.size()`
Acesso	`a[i]`	`lista.get(i)`

Só tipos de objeto: `ArrayList<Integer>` em vez de `<int>`, tal como `Double`, `Boolean`.

Ciclo for-each

```java
for (Animal a : animais) {
  a.fazerSom();
}
```

Lê-se: «para cada animal `a` em `animais`». Funciona com ArrayList e arrays, mas sem índice. Quem precisar de `i` usa o ciclo for normal com `get(i)`.

Durante o for-each não se junta nem se apaga nada: `ConcurrentModificationException`.

Com lambda: `animais.forEach(a -> a.fazerSom());`

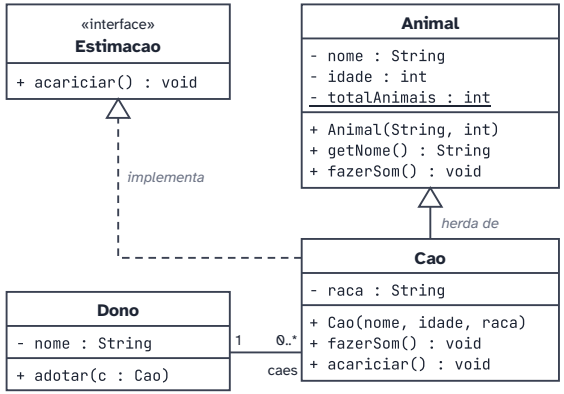
Cábula de Java POO e UML · Programação orientada a objetos

Classes · Objetos · Casos de uso · Sequência

Holger Sebastião · www.sebastiao.org
Versão 1.0 · 2026-10-25

Diagrama de classes

estrutura e relações entre classes



UMA CLASSE: TRÊS COMPARTIMENTOS

em cima	nome, a negrito. Abstrata: *itálico*. Interface: «interface»
ao meio	atributos: `- nome : String`
em baixo	métodos: `+ setIdade(idade : int) : void`

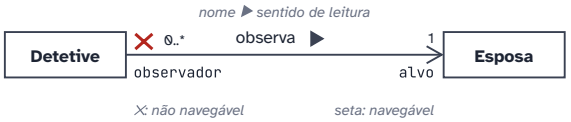
Primeiro a visibilidade, depois o nome, depois **dois pontos e o tipo**, ao contrário do Java. Construtor sem tipo de retorno. `static` fica sublinhado.

`+` public `-` private `#` protected `~` pacote

DO DIAGRAMA AO CÓDIGO

`- nome : String`	`private String nome;`
`+ getNome() : String`	`public String getNome()`
Cao —▷ Animal	`class Cao extends Animal`
Cao ┈▷ Estimacao	`class Cao implements Estimacao`
Dono 1 —— 0..* Cao	`private ArrayList<Cao> caes;`

LER UMA ASSOCIAÇÃO COM RIGOR



Lê-se: «Um detetive **observa** exatamente uma esposa.» **Seta**: o detetive conhece o alvo. ✕: a esposa não pode conhecer o detetive. **Papel** = nome do atributo:

```
private Esposa alvo;  // em Detetive
// em Esposa: nenhum atributo para o detetive
```

RELAÇÕES

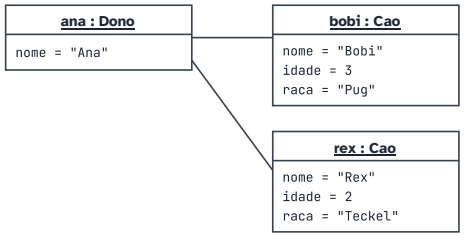
——▷	**Herança** `extends`	«é um». O triângulo aponta para a superclasse.
┈┈▷	**Realização** `implements`	a classe cumpre uma interface.
——	**Associação**	«conhece»: atributo do tipo da outra classe. Seta ——> = só num sentido.
◇——	**Agregação**	«tem»: a parte vive sozinha (Curso ◇—— Aluno).
◆——	**Composição**	«é feito de»: a parte morre com o todo (Casa ◆—— Divisão).
┈┈>	**Dependência**	«usa» de passagem, por exemplo como parâmetro.

MULTIPLICIDADE: QUANTOS?

`1`	exatamente um	`0..1`	no máximo um
`*` = `0..*`	quantos quiser	`1..*`	pelo menos um

Diagrama de objetos

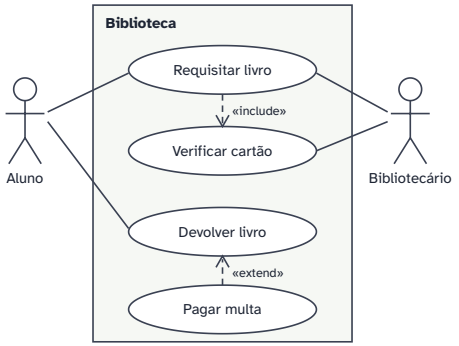
instantâneo



Mostra **objetos** num dado momento: cabeçalho `nome : Classe` sublinhado e, por baixo, atributos **com valores**. Sem métodos, ligações sem multiplicidade.

Diagrama de casos de uso

use case

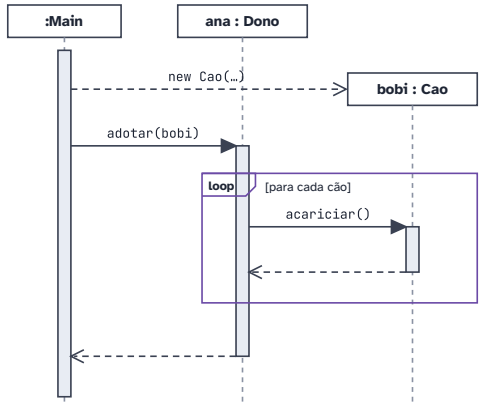


O quê faz o sistema e para quem? **Ator**: pessoa ou sistema de fora. **Caso de uso**: elipse, verbo + nome. Retângulo = fronteira do sistema.

«include»: corre **sempre** também. **«extend»**: só **às vezes**, a seta aponta para o caso base.

Diagrama de sequência

fluxo no tempo



Mostra que objetos enviam que mensagens **uns após outros**. O tempo corre de cima para baixo.

——▶	**mensagem síncrona**: chamada normal de método, quem envia espera
<┈┈┈	**resposta**: regresso, talvez com valor de retorno
——▶	**mensagem assíncrona**: quem envia não espera
┈┈┈▷	**criar objeto** com `new`: a seta acaba no cabeçalho

Linha de vida (tracejada): enquanto o objeto existir. **Barra de ativação**: o objeto está a trabalhar.

`alt`	`[condição]` / `[else]`: como if-else
`opt`	`[condição]`: como if sem else
`loop`	`[condição]`: ciclo

```
public void adotar(Cao c) {
  caes.add(c);
  for (Cao cao : caes) {
    cao.acariciar();
  }
}
```

Visibilidade

modificadores de acesso

JAVA	UML	CLASSE	PACOTE	SUB-CLASSE	TODOS
`public`	`+`	✓	✓	✓	✓
`protected`	`#`	✓	✓	✓	–
nenhum	`~`	✓	✓	–	–
`private`	`-`	✓	–	–	–

protected para atributos que as subclasses usam diretamente. Atenção: o mesmo pacote também lhes acede.

Pacotes

package

```
// Ficheiro: animais/Cao.java
package animais;          // mesmo no topo

public class Cao extends Animal { … }
```

```
// noutro pacote
import animais.Cao;       // uma classe
import animais.*;         // todas de animais
```

Um pacote é uma pasta para classes relacionadas. Nome do pacote = nome da pasta, em minúsculas.

Variáveis de classe

static

```
private static int totalAnimais = 0;
public Animal(String nome, int idade) {
  // … definir os atributos como acima
  totalAnimais = totalAnimais + 1;
}
public static int getTotalAnimais() {
  return totalAnimais;
}
```

`static` pertence à **classe**: `totalAnimais` só existe uma vez para todos os animais. Chama-se pela classe: `Animal.getTotalAnimais()`. UML: sublinhado.

Mensagens de erro típicas

`cannot be applied to given types`: parâmetros errados no `new`

`has private access`: usar getter ou setter

`does not override abstract method`: falta um método da interface

`NullPointerException`: objeto nunca criado com `new`

Cábula de Java POO e UML · Programação orientada a objetos

Associação · Polimorfismo · Todo e parte · Exceções

Holger Sebastião · www.sebastiao.org
Versão 1.0 · 2026-10-25

Associação no código — conhece

0..1 OU 1: UM ATRIBUTO




```
public class Dono {
  private Cao meuCao;        // papel = nome

  public void setMeuCao(Cao cao) {
    this.meuCao = cao;
  }
}
```

0..* OU 1..*: UMA LISTA




```
private ArrayList<Cao> caes = new ArrayList<>();

public void adotar(Cao cao) {
  caes.add(cao);
}
```

NOS DOIS SENTIDOS

Sem seta, ambos se conhecem: cada classe recebe um atributo. Ao ligar, definir os dois lados:

```
ana.setMeuCao(bobi);
bobi.setDono(ana);
```

Polimorfismo com listas

```
ArrayList<Animal> animais = new ArrayList<>();
animais.add(new Cao("Rex", 2, "Teckel"));
animais.add(new Gato("Mia", 5));

for (Animal a : animais) {
  a.fazerSom();   // Au-au, depois Miau
}
```

Numa lista do tipo da superclasse cabem todas as subclasses. É sempre chamado o método do objeto **real**.

```
if (a instanceof Cao) {
  Cao c = (Cao) a;   // verificar, depois cast
}
```

Agregação — tem, a parte vive sozinha




```
public class Curso {
  private ArrayList<Aluno> participantes
      = new ArrayList<>();

  public void inscrever(Aluno a) {
    participantes.add(a);
  }
}

Aluno ada = new Aluno("Ada");
curso.inscrever(ada);
```

A parte é criada **fora** e passada para dentro. Se o curso for apagado, `ada` continua a existir. Um aluno pode estar em vários cursos.

Composição — é feito de, a parte morre com o todo




```
public class Casa {
  private ArrayList<Divisao> divisoes
      = new ArrayList<>();

  public Casa() {
    divisoes.add(new Divisao("Cozinha"));
    divisoes.add(new Divisao("Sala"));
  }
}
```

O todo cria as suas partes **sozinho**, normalmente no construtor. Nenhum outro objeto as recebe. Sem casa, não há divisões. Do lado do losango está sempre `1` (ou `0..1`).

> **Para memorizar:** a parte pode continuar a viver com sentido sem o todo? Sim = agregação ◇, não = composição ◆. Na dúvida, uma associação simples.

Relações num relance

RELAÇÃO	UML	EM JAVA
Herança	——▷	`extends`
Realização	·····▷	`implements`
Associação	——>	atributo do outro tipo
Agregação	◇——	atributo, a parte vem de fora
Composição	◆——	parte criada com `new` no construtor
Dependência	·····>	só parâmetro ou variável local

Exceções — try-catch

Uma exceção é um erro **enquanto** o programa corre. Sem `catch`, o programa para.

```
try {
  // aqui pode correr mal
  int num = Integer.parseInt(texto);
  System.out.println(num);
} catch (NumberFormatException e) {
  // só corre com este erro
  System.out.println("Não é um número!");
} finally {
  // corre sempre (opcional)
}
```

REPETIR A ENTRADA ATÉ ESTAR CERTA

```
import java.util.InputMismatchException;

int idade = 0;
boolean ok = false;
do {
  try {
    idade = entrada.nextInt();
    ok = true;
  } catch (InputMismatchException e) {
    System.out.println("Introduz um número!");
    entrada.nextLine();  // deitar fora o erro
  }
} while (!ok);
```

> ! Uma variável criada dentro do bloco `try` deixa de existir depois. Por isso, `idade` é declarada antes.

ASSINALAR UM ERRO TU PRÓPRIO

```
public void setIdade(int idade) {
  if (idade < 0) {
    throw new IllegalArgumentException(
        "A idade não pode ser negativa");
  }
  this.idade = idade;
}
```

`e.getMessage()` devolve o texto do erro. São permitidos vários blocos `catch`, o mais específico primeiro.