GSI015 – Programação orientada a objetos 1

Slides fornecidos pelo Prof. Renato Pimental

Prof. Bruno Travençolo

1º Semestre – 2021

Sumário

Objetivos

- Ao final do curso, o aluno será capaz de:
  1. Analisar problemas computacionais e propor soluções utilizando conceitos de programação orientada a objetos, como: classes, objetos, herança e polimorfismo;
  2. Desenvolver programas em uma linguagem de programação orientada a objetos.
- Objetivos específicos:
  1. Propiciar transição entre a programação estruturada e a programação orientada a objetos (POO);
  2. Projetar, implementar e testar programas orientados a objetos;
  3. Introduzir conceitos de classes, objetos, herança e polimorfismo;
  4. Apresentar visão geral dos recursos da linguagem Java.

Bibliografia básica

- BOOCH, G. *Object-Oriented Analysis and Design with Applications.* 3.ed. New Jersey: Wesley, 2007 (4).
- BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML, Guia do Usuário.* Rio de Janeiro: Campus, 2000 (5).
- **DEITEL, H. M.; DEITEL P. J. *Java: Como Programar*. 6. ed. Boston: Pearson, 2005 (3).**

Bibliografia complementar

- SIERRA, Kathy; BATES, Bert. *Use a Cabeça – Java*. Editora Alta Books, 2005 (2).
- FOWLER, M. *UML Essencial* (2a Edição). Porto Alegre: Bookman, 2005 (3).
- LARMAN, C. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos*. Porto Alegre: Bookmann, 2001 (19).
- ARNOLD, K.; GOSLING J. *The Java Programming Language*, Addison Wesley, Second Edition. 1996 (8).
- SEDGEWICK R. and WAYNE K., *Introduction to Programming in Java: An Interdisciplinary Approach* . Addison-Wesley, 2008 (3).

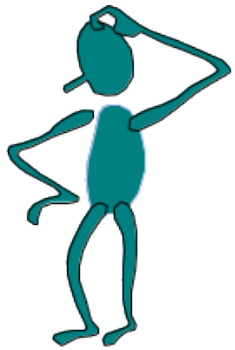
Ementa do curso

- Introdução aos conceitos básicos de POO: classes, objetos, encapsulamento, herança e polimorfismo;
- Conhecimento dos membros que tipicamente compõem classes: construtores, destrutores, variáveis e métodos;
- Entendimento e aplicação dos conceitos de OO em linguagens de programação que suportem tal paradigma;
- Desenvolvimento de sistemas usando POO.

Conteúdo previsto

- Introdução à POO
- Programação procedimental versus POO
- Classes e interfaces
- Objetos, atributos, métodos, construtores e destrutores
- Membros de classe e membros de objetos
- Métodos concretos e abstratos
- Visibilidade e encapsulamento
- Generalização, especialização e herança
- Polimorfismo
- Tratamento de exceções
- Manipulação de arquivos
- GUI

Como *softwares* são construídos?



Linguagens de programação são utilizadas para a construção de programas em computadores

Uma **linguagem de programação** pode ser definida como:

- Conjunto limitado de símbolos e comandos, utilizados para criar programas;
- Método padronizado para expressar instruções para um computador

Por meio dela se estabelece uma comunicação com o computador, fazendo com que ele compreenda e execute o que o programador determinar.

Uma linguagem (natural ou de programação) é formada por:

- **Sintaxe**: a forma ou estrutura das expressões;
- **Semântica**: o significado das expressões.

Sintaxe: determina regras de como se escreve de forma correta em uma linguagem (*regras de escrita*).

Frase sintaticamente correta

Os seguintes países fazem parte do Mercosul: Brasil, Argentina, Paraguai, Uruguai e Venezuela

Frase sintaticamente incorreta

Os serguintes países faz parte do Mercosul: Brasil, Argentina, Paraguai, Uruguai e Venezuela

Considere o comando para a criação e declaração de uma variável, em linguagem Java:

```
int idade;
```

Considere agora o comando para atribuição de valor a uma variável, em linguagem Java:

```
idade = 10;
```

Estes comandos estão sintaticamente corretos, na linguagem de programação Java.

Considere agora os seguintes comandos, também em Java:

```
Int idade;        // ERRO: Int
int idade         // ERRO: Falta ;
```

No caso de atribuição de valores:

```
idade := 10;      // ERRO: :=
idade = 10;       // ERRO: Falta ;
```

Estes comandos estão sintaticamente incorretos, tratando-se especificamente de Java.

Durante o início do aprendizado de uma linguagem de programação, é natural demorar muito tempo procurando erros de sintaxe.

Conforme o programador ganha experiência, entretanto, cometerá menos erros, e os identificará mais rapidamente.

A **Semântica** está relacionada ao significado das palavras ou frases:

Frase semanticamente correta

O Sol é uma estrela

Frase semanticamente incorreta

Os modelos mais sofisticados trazem acentos com abas laterais, volante com ajuste de altura e profundidade e fácil acesso aos pedais.

Considere os comandos, em Java:

```
int idade;      // comandos sintatica e
idade = 10;     // semanticamente corretos
```

Considere agora os seguintes comandos:

```
int idade;
idade = 10.7; // comando de atribuição
    semanticamente incorreto
```

Há **erros de semântica** relacionados ao raciocínio/ lógica de programação

- Para este tipo de erro, o programa executará com sucesso (o computador não irá gerar quaisquer mensagens de erro)
- Mas ele não fará o esperado, apesar de fazer exatamente o que o programador mandar.

- Ocorre em diferentes níveis:
  - Linguagem de máquina;
  - Linguagem de baixo nível;
  - Linguagem de alto nível.
  - Linguagem de muito alto nível.

- Conjunto básico de instruções, em **código binário** (0s e 1s), características de cada computador, correspondentes às suas operações básicas:
  - Instruções lógicas e aritméticas;
  - Instruções para transferência de Informação;
  - Instruções para testes;
  - Instruções de entrada/saída;
- Programação inviável para seres humanos.

Linguagens de baixo nível I

- Linguagem simbólica: bem próxima da **linguagem de máquina**;
- Programação baseada na manipulação de registradores e utilização de interrupções para prover entrada/saída;
- Como há uma correspondência **biunívoca** entre instruções simbólicas e instruções da máquina, as linguagens simbólicas:
  - Dependem do processador utilizado;
  - Permitem escrever programas muito eficientes;
  - Porém: são de utilização muito difícil e sujeitas a erros;

  Exemplo: **Assembly**

Exemplo de programa em Assembly: escrever a mensagem "Olá, mundo" na tela (fonte: https://pt.wikipedia.org/wiki/Programa_Ol%C3%A1_Mundo#Assembly):

```
1     variable:
2     .message    db    "Ola, Mundo!$"
3     code:
4     mov   ah, 9
5     mov   dx, offset  .message
6     int   0x21
7     ret
```

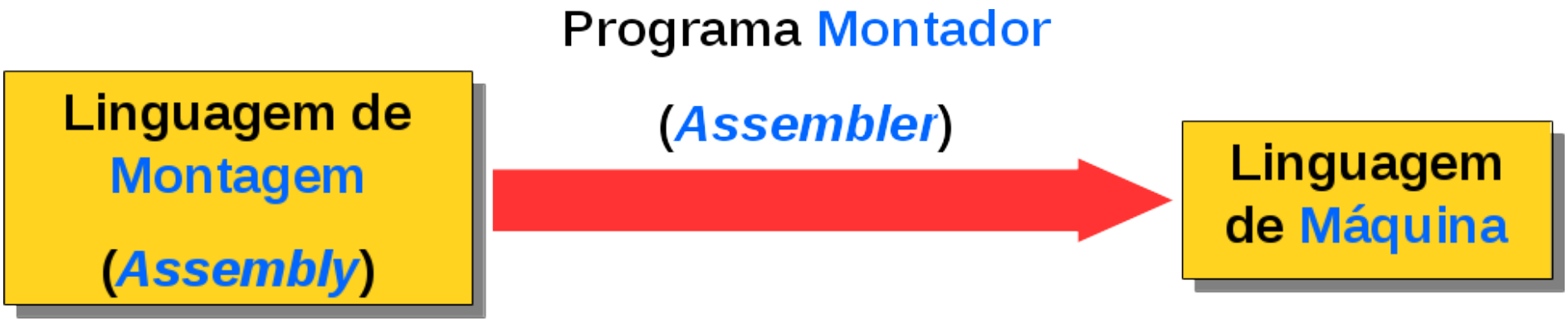
A linguagem Assembly (linguagem de montagem) usa nomes (mnemônicos) e símbolos em lugar dos números:

- Utiliza palavras abreviadas (mnemônicos) indicando operações

```
mov R1, R2
```

- Mnemônico MOV (abreviação de MOVE)
- 2 registradores como parâmetros: `R1` e `R2`
- Processador comanda o movimento do conteúdo de `R2` para `R1` equivalente à instrução Java `R1 = R2;`

A tradução/conversão da linguagem Assembly para a linguagem de máquina se chama **montagem**:



- Programas são escritos usando uma linguagem parecida com a nossa (vocabulário corrente);
- Independe da arquitetura do computador;
- Programação mais rápida e eficiente: mais fácil; programas menos sujeitos a erros;

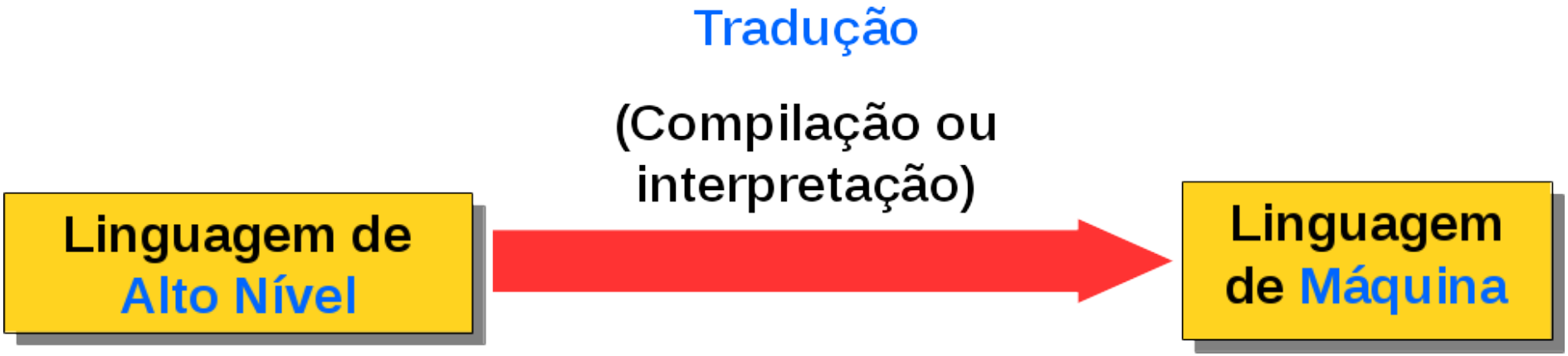
Algumas linguagens de programação e ano em que foram desenvolvidas:

1957	FORTRAN	1972	C	1984	Standard ML
1958	ALGOL	1975	Pascal	1986	C++
1960	LISP	1975	Scheme	1986	CLP(R)
1960	COBOL	1977	OPS5	1986	Eiffel
1962	APL	1978	CSP	1988	CLOS
1962	SIMULA	1978	FP	1988	Mathematica
1964	BASIC	1980	dBase II	1988	Oberon
1964	PL/1	1983	Smalltalk	1990	Haskell
1966	ISWIM	1983	Ada	1995	Delphi
1970	Prolog	1983	Parlog	1995	**Java**

```
1 class HelloWorld
2 {
3     public static void main (String[] args)
4     {
5         System.out.println ("Bem Vindos ao curso de
              POO");
6     }
7 }
```

Os programas escritos em linguagens de alto nível são denominados **código fonte**.

- Os códigos fonte devem ser convertidos para a linguagem de máquina (**tradução**)



As linguagens de muito alto nível têm uma estrutura ainda mais próxima da linguagem humana:

- Definem "o que" deve ser feito, e não "como" deve ser feito
- **Exemplo:** linguagens de consulta a banco de dados, como SQL

Exemplo: SQL, linguagem de consulta para manipular bases de dados.

nome	email	telefone	salario	cargo	*id
João da Silva	jsilva@swhere.com	7363-2334	2300	Gerente	1034
Carlos Ribas	cribas@cblanca.org	8334-2334	1800	Auxiliar	2938
Madalena Reis	mreis@portal.com	6451-5672	2000	Contador	7567
Patricia Horws	phorws@mail.com	4513-6564	2900	Gerente	2314
Carlito Fox	cfox@uol.com.br	5642-7873	1500	Auxiliar	5622
Ricardo Alves	ralves@portal.com	9302-4320	2000	Programador	6762

Apresentar os dados dos campos nome e telefone da tabela Funcionario:

```
SELECT nome, telefone FROM funcionario;
```

Vimos que a conversão de **código fonte** (linguagem de alto nível) para **código executável** (linguagem de máquina) – tradução – é feita de 2 formas:

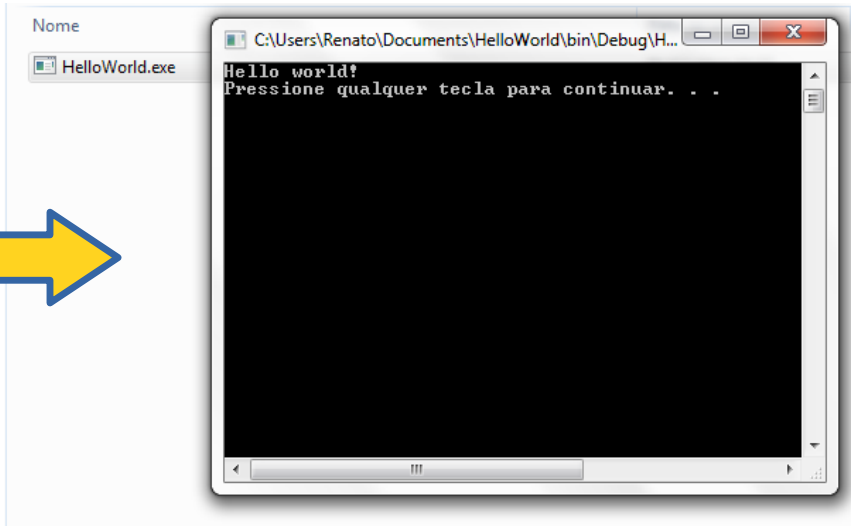
1. compilação
2. interpretação

Vejamos como funciona cada uma destas 2 formas.

- **Análise sintática e semântica**: Programa fonte escrito em uma linguagem qualquer (linguagem fonte) $\Rightarrow$ **programa objeto** equivalente escrito em outra linguagem (linguagem objeto);
- **Compilador**: software tradutor em que a linguagem fonte é uma linguagem de alto nível e a linguagem objeto é uma linguagem de baixo nível (como **assembly**) ou de máquina.

main.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       printf("Ola, mundo!\n");
7       system("pause");
8       return 0;
9   }
10
```



Arquivo .c (Código escrito em linguagem C)

Arquivo executável (.exe) gerado pelo compilador

Figura 1: O compilador

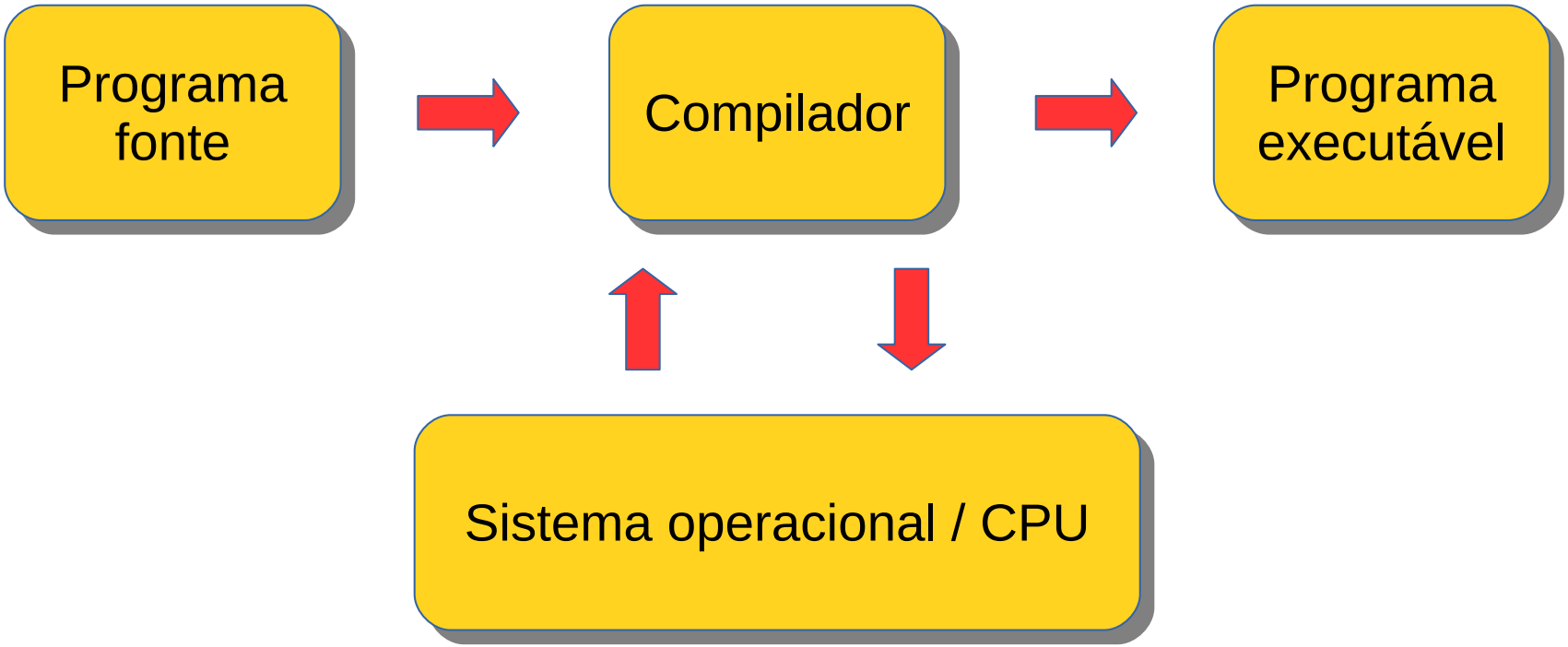


Figura 2: Geração do programa executável

O código executável produzido não é portável

- Diferentes compiladores são construídos para as diferentes arquiteturas de processadores

Exemplos de linguagens **compiladas**:

- C
- Pascal
- FORTRAN
- C++

- É um programa que interpreta (análise sintática e semântica) as instruções do programa fonte **uma a uma**, gerando o resultado;
- Toda vez que o programa é executado, o tradutor transforma cada instrução do código-fonte em uma instrução de código-objeto, que é imediatamente executada:
  - Não é criado todo um programa-objeto, mas apenas a conversão instrução a instrução.

Um interpretador é um programa que executa repetidamente a seguinte sequência:

1. Obter o próximo comando do programa
2. Determinar que ações devem ser executadas
3. Executar estas ações

Caso haja alguma linha de código mal codificada (não respeitando o que foi definido na linguagem), o programa termina sua execução abruptamente em **erro**.

Exemplos de linguagens **interpretadas**:

- HTML
- Haskell
- Lisp

Compilação × interpretação

- Compilação:
  - O programa fonte não é executado diretamente; O programa fonte é convertido em programa objeto e depois é executado; Vantagem: Execução muito mais rápida.
- Interpretação:
  - Executa-se diretamente o programa fonte;
  - Não existe a geração de um módulo ou programa objeto;
  - Vantagem: Programas menores e mais flexíveis.

- Questão: como resolver um determinado problema?
- **Paradigmas de programação**
  - Relacionados à forma de pensar do programador;
  - Como ele busca a solução para o problema;
  - Mostra como o programador analisou e abstraiu o problema a ser resolvido.

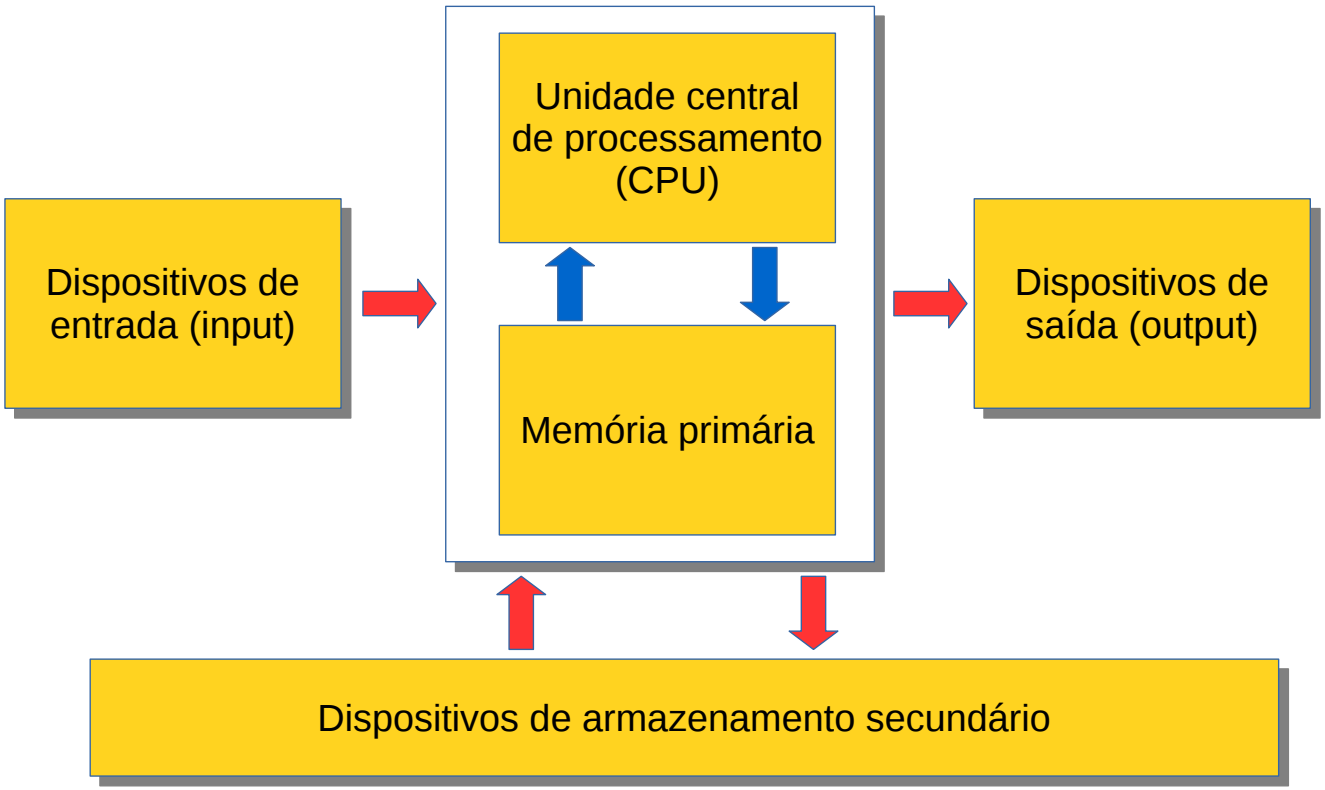
Paradigmas de programação são estilos utilizados pelos programadores para conceber um programa. Um paradigma é um modelo ou padrão conceitual suportado por linguagens que agrupam certas características em comum Respondem de forma diferente a perguntas como:

- O que é um programa?
- Como são modelados e escritos?

- Paradigmas
  - Estruturado;
  - Funcional;
  - Lógico;
  - Orientado a objetos;

Paradigma estruturado I

- Primeiro a surgir, ainda dominante
- Implementado com base na **máquina de von Neumann**



- Também chamado **imperativo**;
- Utilização de 3 estruturas:
  - **Sequencial**;
  - **Condicional**;
  - **Repetição** ou **iterativa**.
- Busca quebrar um problema complexo em partes menores;
- **Programador**:
  - Analisa o problema;
  - Tenta relacionar ações que deverão ser executadas.

O programador descreve a resolução de um problema através de uma série de tarefas elementares (comandos), que o computador pode compreender e executar

```
leia (num1)
leia (num2)
se (num1 > num2) entao
    imprima (num1 é maior que num2)
senao
    imprima (num2 é igual ou maior a num1)
```

Ao final, a sequência de comandos define a resolução do problema A programação é dada por uma sequência de comandos que manipula um volume de dados

Exemplos de linguagens: **Fortran**, C, Basic, Pascal

```
PROGRAM MAIN
INTEGER I, I_START, I_END, I_INC
REAL A(100)

I_START = 1
I_END = 100
I_INC = 1

DO I = I_START, I_END, I_INC

   A(I) = 0.0E0

END DO

END
```

Paradigma funcional I

- Qualquer computação é formulada em termos de funções
- Funções são aplicadas aos parâmetros, e retornam um valor
- As variáveis são os parâmetros formais das funções
- Execução do programa = avaliar funções/expressões

Exemplo da média de uma sequência de números:

```
Divide( Soma(Numeros), Conta(Numeros) )
```

- Estrutura de dados principal: listas
- O paradigma funcional é geralmente utilizado na área de **Inteligência Artificial (IA)**

Exemplos de linguagens: **Lisp**, Haskell , Miranda

```
(defun factorial (N)
    "Compute the factorial of N."
    (if (= N 1)
        1
        (* N (factorial (- N 1)))))
```

Perspectiva de um paradigma baseado no raciocínio: **o que** se quer, em vez de **como** atingi-lo
No paradigma lógico, programar é fornecer dados da seguinte natureza:

- **axiomas**, que indicam alguns fatos sobre o mundo
- regras para derivação de outros fatos (**inferência**)

No modo de programação:

```
homem(socrates).        %Socrates é um homem
mortal(X):-homem(X). % Todos os homens são
                     % mortais
```

No modo de pergunta (execução):

```
?- mortal(socrates).
Yes
```

O programa deve ter um grande volume de informações, denominados de fatos, para que o sistema chegue à solução.

Exemplos de linguagens: **Prolog**, GPSS

```
man(john).
man(adam).

woman(sue).
woman(eve).

married(adam, eve).

married(X) :-
   married(X, _).
married(X) :-
   married(_, X).

human(X) :-
   man(X).
human(X) :-
   woman(X).

% Who is not married?
?- human(X), not married(X).
   X = john ; X = sue
```

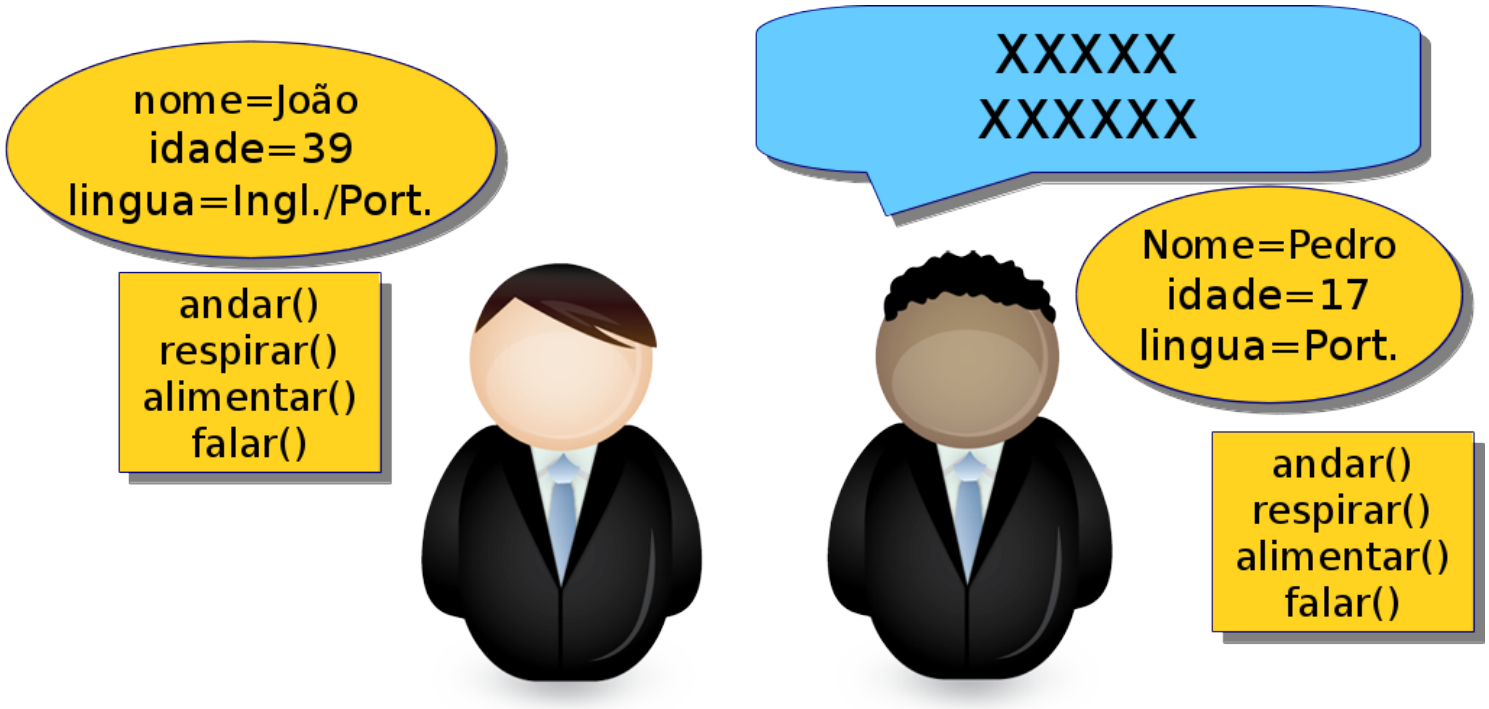
O programa é organizado em função de **objetos**.

Objeto

- Entidade independente com uma identidade e certas características próprias
- Um objeto contém não só as estruturas de dados, mas também as funções que sobre eles agem


nome
idade
lingua
andar()
respirar()
alimentar()
falar()
nome
idade
lingua
andar()
respirar()
alimentar()
falar()

A comunicação entre diferentes objetos ocorre por meio de **trocas de mensagens**:



Programação estruturada:

- Preocupação maior é com as **estruturas de controle** (como construir programas usando as estruturas de sequência, seleção e repetição)
- Preocupa-se com os **algoritmos** e cada módulo que compõe um programa é um algoritmo específico

Programação orientada a objetos:

- Preocupação maior é com os **dados** que o programa irá tratar.
- Preocupa-se com as estruturas de dados e com as operações que podem ser executadas os dados.
- Cada estrutura de dados e suas operações é denominada **classe**.
- Cada módulo que compõe um programa é uma classe.

Exemplo de programação estruturada em C

```c
#include <stdio.h>
#include <stdlib.h>

// Define um novo tipo de dados
typedef struct {
   int num, den;
} fracao;

int mdc(int a, int b) { // Podem existir
      operacoes sobre outros tipos de dados
   int r;
   while (b>0) {
      r = a%b;
      a = b;
      b = r;
   }
   return a;
}

// cada possivel operacao sobre o tipo de
      dados e uma funcao independente
fracao simplificar(fracao a) {
   int m;
   m = mdc(a.num, a.den);
   a.num = a.num/m;
   a.den = a.den/m;
   return a;
}

fracao somar(fracao a, fracao b) {
   fracao c;
   c.num = a.num*b.den + a.den*b.num;
   c.den = a.den*b.den;
   c = simplificar(c);
   return c;
}

// A manipulacao de dados do novo tipo pode
      ser feita em qualquer funcao do
      programa
int main(int argc, char *argv[])
{
   fracao a,b,c;
   a.num = 5;
   a.den = 3;
   b.num = 2;
   b.den = 7;
   c= somar(a,b);
   printf("c = %d/%d\n", c.num, c.den);
   return 0;
}
```

Exemplo de programação OO em Java I

```java
// arquivo: Fracao.java
// A classe Fracao define a estrutura de
      dados e as operacoes possiveis sobre
      essa estrutura
public class Fracao
{
   private int den;
   private int num;

   public Fracao() {
      num = 0;
      den = 1;
   }

   public Fracao(int a, int b) {
      num = a;
      den = b;
   }

   private void simplificar() {
      int r,x,y;
      x = num;
      y = den;
      while (y>0) {
         r = x%y;
         x = y;
         y = r;
      }
      num = num/x;
      den = den/x;
   }

   public void somar(Fracao a, Fracao b) {
      den = a.den*b.den;
      num = a.num*b.den + b.num*a.den;
      simplificar();
   }

   public void mostrar() {
      System.out.println(num + "/" + den);
   }
}
```

Exemplo de programação OO em Java II

```java
// arquivo: Exemplo.java
// A classe Exemplo constroi OBJETOS da CLASSE Fracao e USA tais
     objetos como desejar.
public class Exemplo
{
    public static void main(String[] args)
    {
        Fracao a = new Fracao(5,3);
        Fracao b = new Fracao(2,7);
        Fracao c = new Fracao();
        c.somar(a,b);
        c.mostrar();
    }
}
```

Comparativo: Estruturado vs. OO

Na programação orientada a objetos, o principal conceito é o de **objeto**.

Programação estruturada

```
int x,y;
fracao a,b;

x = 5;
a.num = 3;
a.den = 2;
```

x e y são **variáveis** do tipo primitivo `int`, ou seja, exemplares do tipo `int`. a e b são variáveis de um **novo tipo** `Fracao`, definido pelo usuário, ou seja, exemplares do tipo `Fracao`.

Programação orientada a objetos

```
int x,y;
Fracao a,b;

x = 5;
a = new Fracao(3,2);
```

a e b são **objetos** de uma **nova classe** `Fracao`, definida pela usuário, ou seja, **instâncias** da classe `Fracao`. Uma classe define **todas** as **operações** (**métodos**) possíveis sobre seus objetos.

Como fazer um programa a partir da especificação de um problema?

Exemplo: Desenvolva um programa que, dado um aluno, calcule a média das provas da disciplina de Programação Orientada a Objetos (POO), conforme critérios apresentados em aula.

Como você faria para resolver o problema?

Algo como:

- Identificar os **dados** de entrada;
- Identificar que **resultados** o programa deve fornecer;
- Escrever uma **sequência de instruções** (**comandos**) para *manipular* os dados de entrada e gerar os resultados esperados.

Uma especificação mais completa:

Desenvolva um sistema para fazer a avaliação da disciplina de POO. O sistema deverá receber as notas de todas as avaliações realizadas pelos alunos (provas e projeto), bem como suas frequências e deverá gerar um relatório com as seguintes informações: nota final e situação de cada aluno, menor e maior nota da turma, nota média da turma, total de alunos aprovados, reprovados e de exame.

- **Para problemas maiores:** divide-se o problema em **funções** (procedimentos, sub-rotinas) com objetivos claros e uma interface bem definida com outras funções
- **Para problemas ainda maiores:** agrupam-se funções relacionadas em **módulos** (possivelmente em arquivos diferentes)

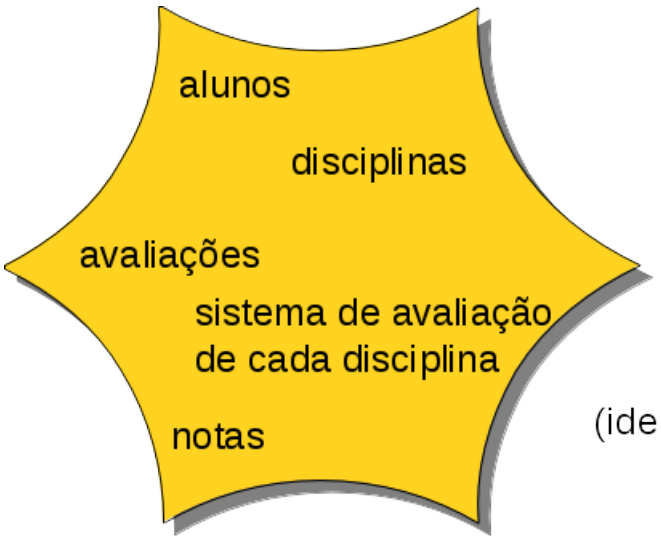
Programação estruturada

Desenvolvimento *top-down* (a. parte-se de uma solução inicial geral e vai decompondo o problema; b. resolvem-se os problemas menores).

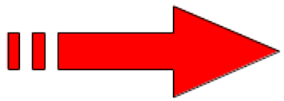
Mas ainda assim as funções e módulos são **listas de instruções** para manipular **dados**.

Em programação estruturada, os programas são uma sequência de comandos (ordens) para o computador executar.

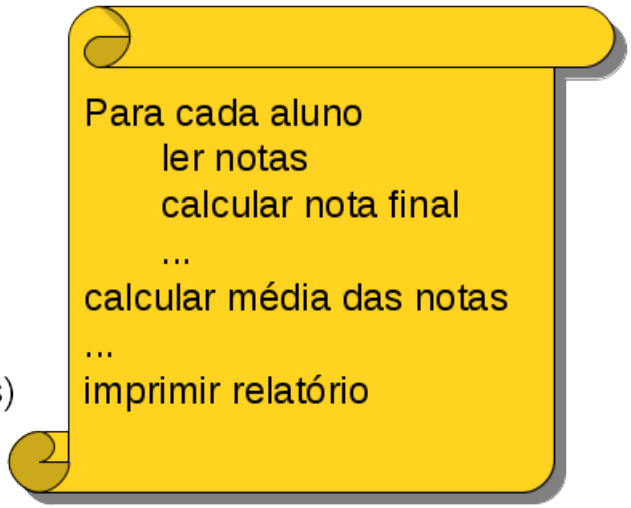
Modelagem I



Espaço do problema
(lugar onde o problema existe)
ex: mundo real



Mapeamento
(identificação de abstrações)



Espaço da solução
(lugar onde será gerada a solução)
ex: computador

A construção de um software envolve um processo de **mapeamento** de objetos pertencentes ao **espaço de problemas** para o **espaço de soluções**.
De maneira que operações sobre essas representações abstratas correspondam a operações do mundo real.

Quanto mais próximo (conceitualmente) o espaço de soluções estiver do espaço de problemas, mais fácil será:

- O desenvolvimento da aplicação;
- Assegurar a compreensão, confiabilidade e manutenção da aplicação.

No **mundo real** lidamos com **objetos**, como *pessoas*, *carros*, *celulares*, *computadores*, ...

Tais objetos não são como **funções**:

Um **objeto real** tem ambos:

- **Características** (atributos):
  - Pessoa: cor dos olhos, idade, nome, CPF, salário, ...
  - Carro: potência, número de portas, ano, modelo, ...
- **Comportamento** – ex.: resposta do objeto a algum estímulo:
  - Ao tocar o telefone, a pessoa atende;
  - Quando o freio é pressionado, o carro para.

Em programação estruturada, as funções trabalham sobre os dados, mas não têm uma ligação íntima com os mesmos.

A **programação orientada a objetos** é uma forma de pensar um problema utilizando conceitos do **mundo real**, e não conceitos computacionais:

- Objetos do mundo real são mapeados em objetos no computador;
- Funções e dados estão juntos, formando um objeto.

Etapas da modelagem:

- **Abstrair**: formular conceitos generalizados extraindo características comuns de exemplos específicos, descartando detalhes que não são importantes
- **Modelar**: criar um modelo que represente o objeto do mundo real

Exemplo da avaliação em POO:

- Que **objetos** podemos identificar?
- Existem grupos de objetos semelhantes (**classes** de objetos)?
- Quais são as características (**atributos**) de cada um?
- Quais são os possíveis **comportamentos** dos objetos?
- Como eles **interagem**?

Vantagens:

1. A modelagem do problema é mais simples;
2. Pode-se trabalhar em um nível mais elevado de abstração;
3. Maior facilidade para reutilização de código;
4. Maior facilidade de comunicação com os usuários e outros profissionais;
5. Redução das linhas de código programadas;
6. Separação das responsabilidades (classes e objetos podem ser desenvolvidos independentemente);
7. Maior flexibilidade do sistema e escalabilidade;
8. Facilidade de manutenção.

Linguagens para programação OO são linguagens que têm facilidades para o desenvolvimento de programas OO.

- Mas o conceito depende mais da mentalidade do programador do que da linguagem de programação utilizada.
- É possível ter:
  - programas razoavelmente OO em linguagens imperativas;
  - programas estruturados em linguagens OO.

POO trata da organização geral do programa e não de detalhes de implementação.

Algumas linguagens OO:

1. **SIMULA**
   - Final da década de 60;
   - Codificar simulações;
   - Primeira linguagem que implementava alguns conceitos de OO, como classes e herança.
2. **SMALLTALK**
   - Linguagem criada pela Xerox entre as décadas de 70 e 80;
   - Primeira linguagem orientada a objetos;
   - Ideias de SIMULA + princípio de objetos ativos, prontos a reagir a mensagens que ativam comportamentos específicos do objeto;
   - Pode ser considerada a linguagem OO mais pura.

3. C++
   - Década de 80;
   - Adaptar os conceitos da OO à linguagem C;
   - É uma linguagem híbrida (multi-paradigma: imperativo, OO, programação genérica).
4. Java
   - Desenvolvida nos anos 90
   - Popularizou a Orientação a Objetos
   - Baseada no C++
   - Linguagem altamente portável
   - Grande interação com a Web

Considere a especificação de notas de POO apresentada em slides anteriores. Faça um programa em uma linguagem estruturada (ex. C) para solucioná-lo.

Identifique os passos necessários à modelagem desse problema segundo o paradigma estruturado.

Referências

Os slides dessa apresentação foram cedidos por:

- Prof. M. Zanchetta do Nascimento, FACOM/UFU
- Profa. Katti Faceli, UFSCar/Sorocaba
- Prof. José Fernando R. Junior, ICMC-USP/São Carlos
- Prof. Senne, Unesp/Guaratinguetá
- Prof. Augusto Chaves, UNIFESP/SJCampos

Outras referências usadas: Apostilas de POO em:
http://www.riopomba.ifsudestemg.edu.br/dcc/dcc/materiais/1662272077_POO.pdf
http://www.jack.eti.br/www/arquivos/apostilas/java/poo.pdf
SEBESTA, ROBERT W., Conceitos de Linguagens de Programação, 5a ed., Bookman, 2003

Objeto

Um elemento ou entidade do mundo real de um determinado domínio

Exemplos:

- *Objetos físicos*: livro, mesa, pessoa, mercadoria, etc.
- *Ocupações de pessoas*: cliente, vendedor, etc.
- *Eventos*: compra, telefonema, etc.
- *Lugares*: loja, cidade, etc.
- *Interações entre objetos*: item (de uma nota fiscal), parágrafo (em um sistema editor de texto), etc.

Objetos são instâncias de classes:

- As **classes** é que determinam quais informações o objeto contém, e como manipulá-las.

Objetos podem reter um estado (informação) e possuem operações para examinar ou alterar seu estado.

É através dos objetos que praticamente todo o processamento ocorre em sistemas OO

Objeto III

Exemplo: objeto **cachorro**

Características próprias, como:

- Um nome;
- Uma idade;
- Um comprimento de pêlos;
- Uma cor dos pêlos;
- Uma cor dos olhos;
- Um peso;
- ...

As características que descrevem um objeto são denominadas **atributos**.

Exemplo: objeto **cachorro**

Pode executar ações, como:

- Latir;
- Correr;
- Sentar;
- Pegar uma bola;
- Comer;
- Dormir;
- ...

As ações que um objeto pode executar são denominadas **métodos**.

A única forma de interação com os objetos é através de seus métodos:

- Para alimentar o cachorro Lulu, usamos o método **Comer**
- Para brincar com ele, usamos o método **Pegar uma Bola**
- etc.

O conjunto de métodos disponíveis em um objeto é chamado **interface**.

Atributos

Como visto, objetos do mundo real têm *propriedades*:

- Essas propriedades recebem o nome de **atributos**. São como *variáveis* ou *campos* que armazenam os valores das características dos objetos

Exemplo (Cachorro):

- *Nome*: Lulu
- *Idade*: 2 anos
- *Comprimento de pêlos*: curto
- *Cor dos pêlos*: marrom
- *Cor dos olhos*: marrom
- *Peso*: 4 kg

Estado de um objeto: conjunto de valores de seus atributos em um determinado instante

- Para haver mudança de valores, são necessários estímulos internos ou externos

Estado (caso anterior):

- *Nome*: Lulu
- *Idade*: 2 anos
- *Comprimento de pêlos*: curto
- *Cor dos pêlos*: marrom
- *Cor dos olhos*: marrom
- *Peso*: 4 kg

Dois cachorros diferentes:

Estado

- *Nome*: Lulu
- *Idade*: 2 anos
- *Comprimento de pêlos*: curto
- *Cor dos pêlos*: marrom
- *Cor dos olhos*: marrom
- *Peso*: 4 kg

Estado

- *Nome*: Rex
- *Idade*: 4 anos
- *Comprimento de pêlos*: longo
- *Cor dos pêlos*: branco
- *Cor dos olhos*: preto
- *Peso*: 10 kg

Métodos são os procedimentos ou funções que realizam as ações do objeto, ou seja, implementam as ações que o objeto pode realizar.

É por seus métodos que um objeto se manifesta e através deles que o objeto interage com outros objetos.

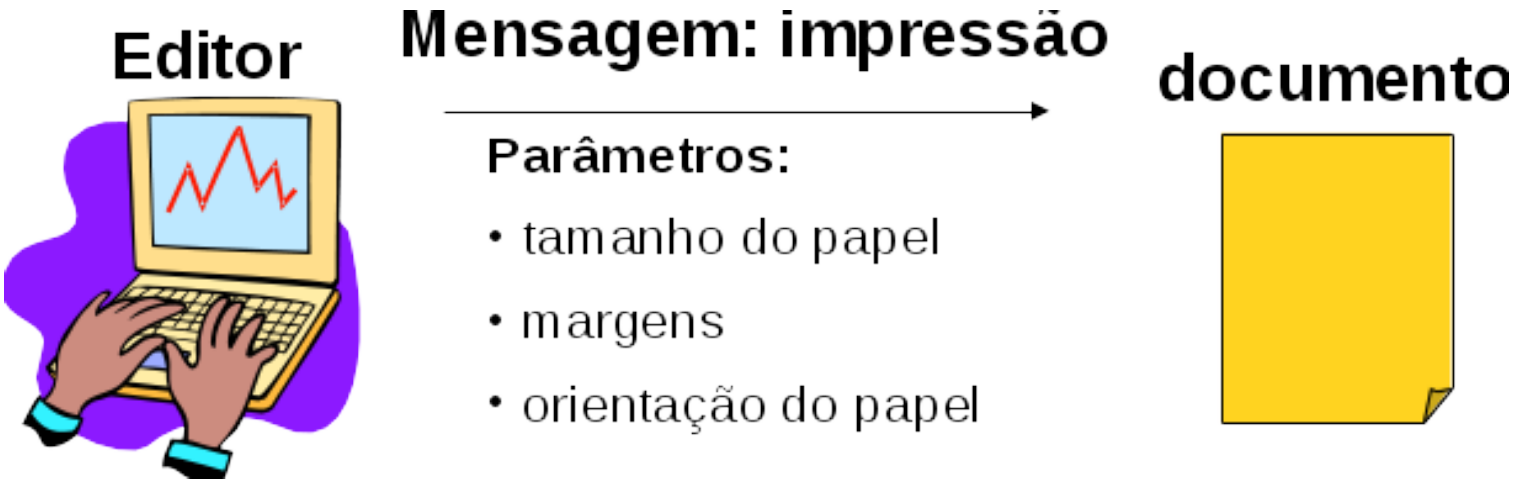
Comportamento de um objeto: como ele age e reage em termos de mudanças de estado e trocas de mensagens com outros objetos

- Execução de uma operação (um método)
- Métodos são responsáveis pelas ações, que podem envolver acessar ou alterar os valores dos atributos

Se a requisição é feita pelo outro objeto, ela é enviada por uma **mensagem**.

- Mensagem é solicitação para que objeto execute um método.

Quem envia a mensagem não necessita saber como o receptor irá tratá-la: Deve apenas *conhecer o resultado final* do tratamento e *o que é necessário para obtê-lo*.



Um **objeto** não é muito diferente de uma variável normal:

- Ex. quando se define uma variável do tipo `int` em Java, essa variável tem:
  - Um espaço em memória para registrar o seu estado (valor);
  - Um conjunto de operações que podem ser aplicadas a ela (operadores que podem ser aplicados a valores inteiros).
- Quando se cria um **objeto**, ele adquire:
  - Um espaço em memória para armazenar seu estado (valores de seus atributos);
  - Um conjunto de operações que podem ser aplicadas ao objeto (seus métodos).

Identificar atributos e métodos em:

1. Uma tela de computador

Exercício II

Identificar atributos e métodos em:

1. Uma tela de computador
   - Atributos:
     - `Modo de operação (texto, gráfico);`
     - `Tamanho horizontal;`
     - `Tamanho vertical;`
     - `Paleta de cores.`
   - Métodos:
     - `modo texto ( );`
     - `modo gráfico ( );`
     - `muda cor ( );`
     - `escreve caractere ( );`
     - `muda dimensões (x,y).`

Defina objetos, atributos e métodos do seguinte sistema:

Uma biblioteca necessita de um sistema que atenda a:

- Cadastro de usuários (professores, alunos ou funcionários), com endereço completo;
- Cadastro de obras, com sua classificação (livros científicos, periódicos científicos, periódicos informativos, periódicos diversos, entretenimento), língua, autores, editoras, ano e mídia onde se encontra o exemplar da obra.

Uma **classe** representa um *conjunto de objetos* que possuem características e comportamentos comuns.

- Um **objeto** é uma *instância* de uma **classe**;
- Ou seja, criam-se objetos baseados no que é definido nas classes.

Ênfase na realidade deve ser nas **classes**.

Exemplo: Cachorros podem ser descritos pelos mesmos atributos e comportamentos, pois são da **mesma classe**:

Cachorro_1

- *Nome*: Lulu
- *Idade*: 2 anos
- *Comprimento de pêlos*: curto
- *Cor dos pêlos*: marrom
- *Cor dos olhos*: marrom
- *Peso*: 4 kg

Cachorro_2

- *Nome*: Rex
- *Idade*: 4 anos
- *Comprimento de pêlos*: longo
- *Cor dos pêlos*: branco
- *Cor dos olhos*: preto
- *Peso*: 10 kg

Classe Cachorro

Objetos da mesma possuem a mesma definição para métodos e atributos (embora valores sejam, em geral, diferentes).

Exemplo 2: Classe gato, formada por objetos "gato"

Algumas características:

- Nome
- Idade
- Comprimento dos pêlos
- Cor dos pêlos
- Peso

Algumas ações:

- Miar
- Comer
- Dormir
- Subir na árvore

Há atributos e métodos comuns entre cães e gatos. O que fazer? Criar a *super-classe* `Mamíferos`. Veremos mais a respeito quando estudarmos o conceito de **herança**.

- HORSTMANN, Cay S.; CORNELL, Gary. *Core Java 2: Vol.1 – Fundamentos*, Alta Books, SUN Mircosystems Press, 7a. Edição, 2005.
- DEITEL, H. M.; DEITEL, P. J. *JAVA – Como Programar*, Pearson Prentice-Hall, 6a. Edição, 2005.
- https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

- Aplicativos ou softwares Java são criados baseados na **sintaxe** do Java.
- Os programas em Java são escritos baseados na criação de classes.
- Cada classe segue as estruturas de sintaxe do Java, definida pelo agrupamento de **palavras-chave**, ou **palavras reservadas**, e nomes de classes, atributos e métodos.

Palavras reservadas do Java:

abstract	boolean	break	byte
case	catch	char	class
continue	default	do	double
else	extends	false	final
finally	float	for	if
implements	import	instanceof	int
interface	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	void	volatile	while

Palavras reservadas, mas não usadas pelo Java: `const`, `goto`

```
1  public class media {
2      public static void main(String[] args) {
3          //variaveis do tipo real
4          float nota1, nota2, nota3, nota4,
               mediaAritmetica;
5          //entrada de dados
6          Scanner entrada = new Scanner (System.in);
7          System.out.println("Entre com a nota 1: ");
8          nota1= entrada.nextFloat ();
9          System.out.println("Entre com a nota 2: ");
10         nota2= entrada.nextFloat();
11         System.out.println("Entre com a nota 3: ");
12         nota3= entrada.nextFloat();
13         System.out.println("Entre com a nota 4: ");
```

```
14          nota4= entrada.nextFloat();
15          //processamento
16          mediaAritmetica = (nota1+nota2+nota3+nota4)/4;
17          //resultados
18          System.out.printf ("A média aritmética: %.2f",
                mediaAritmetica);
19          if (mediaAritmetica >= 7.0){
20              System.out.printf("Aluno aprovado!");
21          }//fim do if
22      } //fim do método main
23  } //fim da classe média
```

Declarando classes I

Uma classe é declarada com a palavra reservada `class`, seguida do nome da classe e de seu corpo *entre chaves* (como em C).

```
1 <modificador de acesso> class NomeDaClasse
2 {
3     // declaracao dos atributos
4     // declaracao dos métodos
5 }
```

O nome da classe é também chamado de **identificador**.

Regras para nome de uma classe:

- Não pode ser uma palavra reservada do Java
- Deve ser iniciado por uma letra, ou _ ou $
- Não pode conter espaços

```
public class Pessoa
{
    //declaracao dos atributos
    //declaracao dos métodos
}
```

Sugestão de estilo para nomes de classes (boa prática de programação):

- A palavra que forma o nome inicia com letra maiúscula, assim como palavras subsequentes:
  - Exemplos: `Lampada`, `ContaCorrente`, `RegistroAcademico`, `NotaFiscalDeSupermercado`, `Figura`, ...
- Devem preferencialmente ser **substantivos**
- Sugere-se que cada classe em Java seja gravada em um **arquivo separado**, cujo nome é o nome da classe seguido da extensão .java

Ex.: classe Pessoa no arquivo **Pessoa.java**

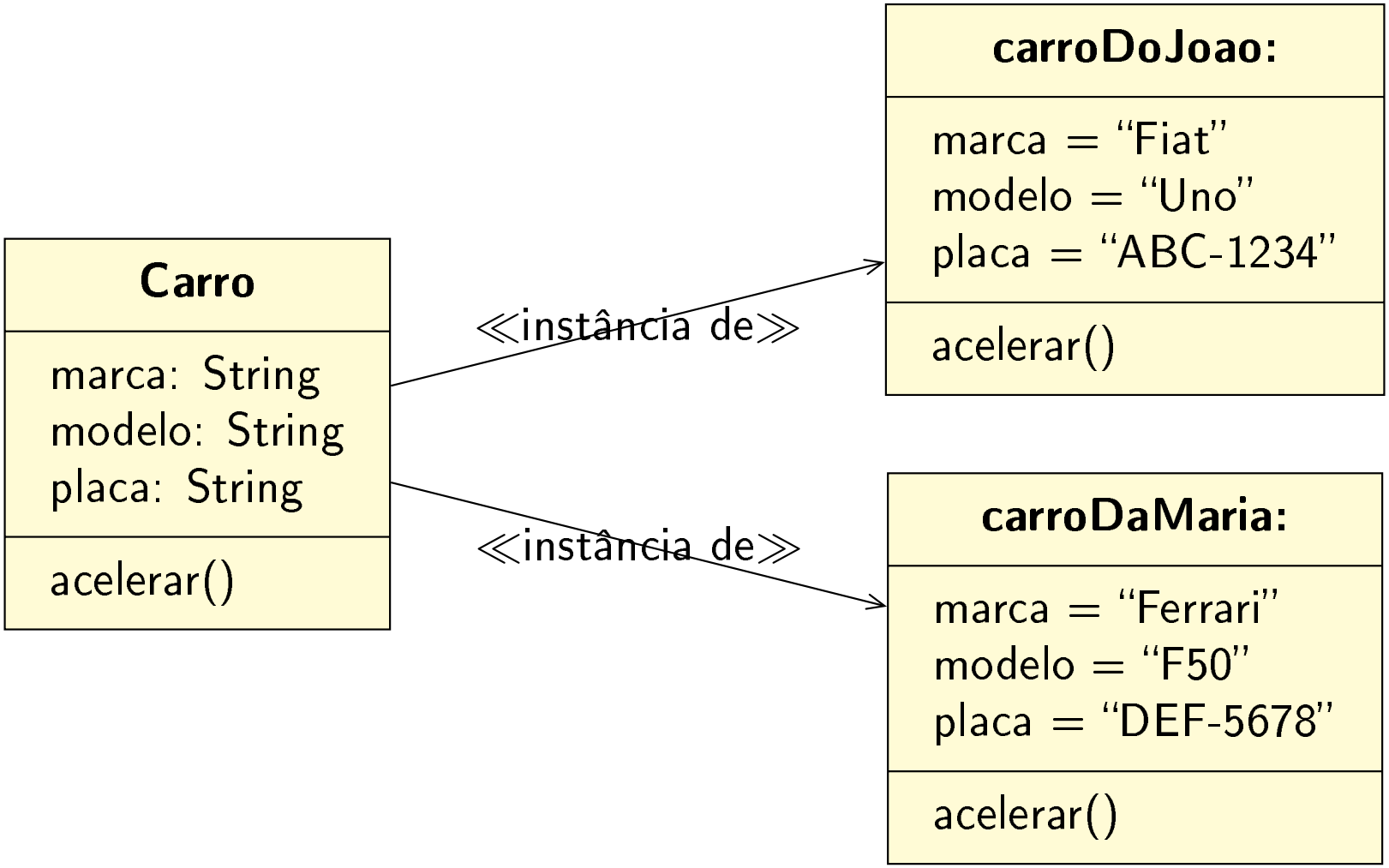
```
1 public class Pessoa {
2     //declaracao dos atributos
3     //declaracao dos métodos
4 }
```

Atributos / variáveis de instância I

- **Atributos / variáveis de instância**: espaço de memória reservado para armazenar dados, tendo um nome para referenciar seu conteúdo;
- Um **atributo** ou **variável de instância** (ou ainda **campo**) é uma variável cujo valor é específico a cada objeto;
- Ou seja, cada objeto possui uma cópia particular de atributos/variáveis de instância com seus próprios valores.
- **Estilo de nome (Java)**: "Camel Case" com primeira letra da primeira palavra minúscula, primeira das demais maiúscula. **Ex.:** `nome`, `dataDeNascimento`, `caixaPostal`.

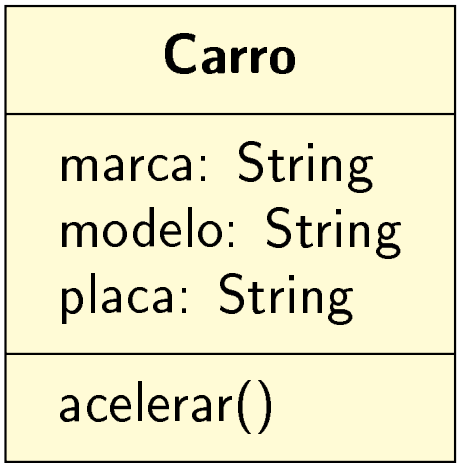
Exemplo: classe `Carro`

Atributos / variáveis de instância III



Atributos / variáveis de instância IV

```
public class NomeDaClasse
{
    //variáveis de instâ
       ncia
    //métodos
}
```



- Variáveis de instância são definidas dentro da classe, fora dos métodos;
- Inicializadas automaticamente.
  - Ou por meio de **construtores** dos objetos da classe.

```
public class Carro
{
    String marca;
    String modelo, placa;
    // métodos ...
}
```

Tipos de dados I

- **Tipos**: representam um valor ou uma coleção de valores, ou mesmo uma coleção de outros tipos.
- Tipos **primitivos**:
  - Valores são armazenados nas variáveis diretamente;
  - Quando atribuídos a outra variável, valores são copiados.
- Tipos **por referência**:
  - São usados para armazenar referências a objetos (localização dos objetos);
  - Quando atribuídos a outra variável, somente a referência é copiada (não o objeto).

Tipos de dados II

Tipos primitivos:

- Números inteiros:

Tipo	Descrição	Faixa de valores
`byte`	inteiro de 8 bits	-128 a 127
`short`	inteiro curto (16 bits)	-32768 a 32767
`int`	inteiro (32 bits)	-2147483648 a 21474863647
`long`	inteiro longo (64 bits)	-2^{63} a $2^{63}-1$

- Números reais (IEEE-754):

Tipo	Descrição	Faixa de valores
`float`	decimal (32 bits)	$-3,4e{+}038$ a $-1,4e{-}045$; 1,4e−045 a 3,4e+038
`double`	decimal (64 bits)	$-1,8e{+}308$ a $-4,9e{-}324$; 4,9e−324 a 1,8e+308

Tipos de dados III

- Outros tipos

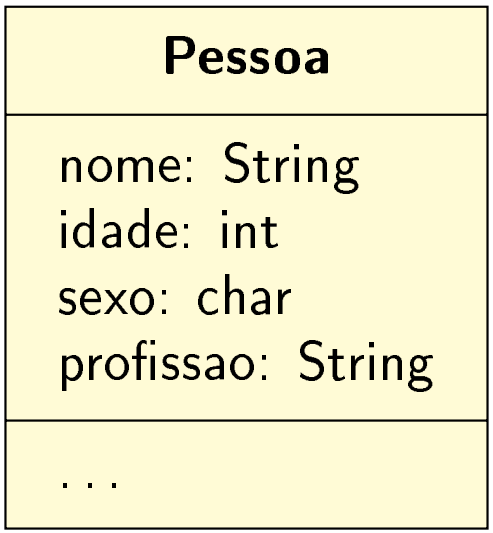
Tipo	Descrição	Faixa de valores
`char`	um único caractere (16 bits)	`'\u0000'` a `'uFFFF'` (0 a 65535 Unicode ISO)
`boolean`	valor booleano (lógico)	`false`, `true`

Tipos **por referência** (ou não-primitivos)

- Todos os tipos não primitivos são tipos por referência (ex.: as próprias classes)
- *Arrays* e *strings*:
  - Ex.: `String teste = "UFU Sta. Monica";`
  - Ex.: `int []v = {3,5,8};`

Criar um modelo em Java para uma pessoa. Considere os atributos `nome`, `idade`, `sexo` e `profissão` (não se preocupe ainda com o comportamento/métodos).

Modelagem:



```java
public class Pessoa
{
    String nome;
    int idade;
    char sexo;
    String profissao;
// métodos ...
}
```

- Os **métodos** implementam o *comportamento* dos objetos;
- Um método agrupa, em um bloco de execução, uma sequência de comandos que realizam uma determinada função;
- Os métodos em Java são declarados dentro das classes que lhes dizem respeito;
- Eles são capazes de **retornar informações** quando completam suas tarefas.
- **Estilo de nome (Java)**: "Camel Case" igual à forma usada para atributos. Ex.: `salvar`, `salvarComo`, `abrirArquivo`.

Sintaxe da declaração de um método (semelhante a C):

```
1  public class NomeDaClasse
2  {
3      // declaração dos atributos
4      // ...
5
6      <mod. acesso> <tipo de retorno> nomeDoMétodo (
           argumento[s])
7      {
8          // corpo do método
9      }
10 }
```

Exemplo:

```
1  public class Carro
2  {
3      // declaração dos atributos
4      // ...
5
6      public void acelerar()
7      {
8          // corpo do método
9      }
10 }
```

- Em geral, um método recebe argumentos / parâmetros, efetua um conjunto de operações e retorna algum resultado.
- A definição de método tem cinco partes básicas:
  - modificador de acesso (`public`, `private`, ...);
  - nome do método;
  - tipo do dado retornado;
  - lista de **parâmetros** (argumentos);
  - corpo do método.

- Os métodos aceitam a passagem de um número determinado de parâmetros, que são colocados nos parênteses seguindo ao nome;
- Os argumentos são **variáveis locais** do método – assim como em C.

Exemplo:

```
float hipotenusa (float catetoA, float catetoB)
{
    // corpo do método
    // ...
}
```

Se o método não utiliza nenhum argumento, parênteses vazios devem ser incluídos na declaração.

Exemplo:

```
public class Carro
{
    String marca;
    String modelo;
    String placa;
    public void acelerar()
    { // corpo do método
    }
}
```

Um método pode devolver um valor de um determinado tipo de dado.

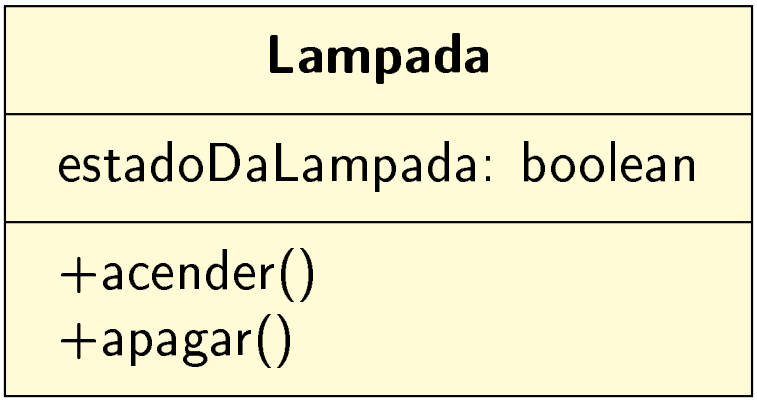
- Nesse caso, existirá no código do método uma linha com uma instrução `return`, seguida do valor ou variável a devolver.

```
1 class Administrativo {
2     float salario;
3     public float retorneSalario( )
4     {
5         //calcular salario
6         return salario;
7     }
8 }
```

Se o método não retorna nenhum valor, isto deve ser declarado usando-se a palavra-chave `void` – como em C.
Exemplo:

```
public class Carro {
    String marca;
    String modelo;
    String placa;
    public void acelerar() // void indica que nao ha
        retorno de valor
    {
    }
}
```

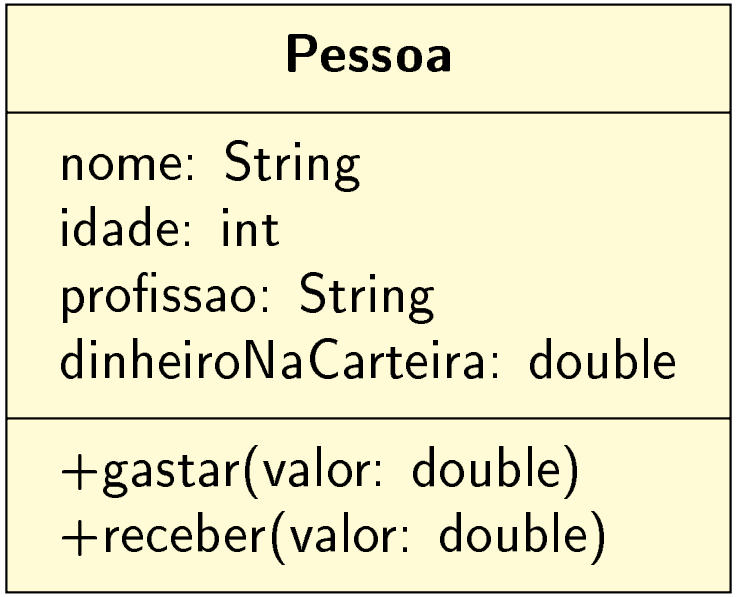
Criar um modelo em Java para uma lâmpada. As operações que podemos efetuar nesta lâmpada também são simples: podemos ligá-la ou desligá-la. Quando uma operação for executada, imprima essa ação.



Exercício – métodos II

```
1  public class Lampada {
2      boolean estadoDaLampada;
3
4      public void acender( )
5      {
6          estadoDaLampada = true;
7          System.out.println("A acao acender foi executada");
8      }
9      public void apagar( )
10     {
11         estadoDaLampada = false;
12         System.out.println("A acao apagar foi executada");
13     }
14 }
```

Criar um modelo em Java para uma pessoa. Considere os atributos nome, idade e profissão. A pessoa pode gastar ou receber uma certa quantia de dinheiro (assuma que o dinheiro se encontra na carteira).



```java
public class Pessoa
{
    String nome, profissao;
    int idade;
    double dinheiroNaCarteira;
    public void gastar( double valor )
    {
        dinheiroNaCarteira = dinheiroNaCarteira - valor;
    }
    public void receber( double valor )
    {
        dinheiroNaCarteira = dinheiroNaCarteira + valor;
    }
}
```

- **Declaração**: a seguinte instrução declara que a variável `nomeDoObjeto` refere-se a um objeto / instância da classe `NomeDaClasse`:

```
NomeDaClasse nomeDoObjeto;
```

- **Criação**: a seguinte instrução cria (em memória) um novo objeto / instância da classe `NomeDaClasse`, que será referenciado pela variável `nomeDoObjeto` previamente declarada:

```
nomeDoObjeto = new NomeDaClasse();
```

As duas instruções acima podem ser combinadas em uma só:

```
NomeDaClasse nomeDoObjeto = new NomeDaClasse();
```

```
public class Carro {
    String marca;
    String modelo;
    String placa;
    public void acelerar()
    {
    }
}
```

```
...
Carro carro1 = new Carro();
...
```

O comando `new` cria uma instância de uma classe:

- Aloca espaço de memória para armazenar os atributos;
- Chama o **construtor** da classe;
- O construtor inicia os atributos da classe, criando novos objetos, iniciando variáveis primitivas, etc;
- Retorna uma *referência* (**ponteiro**) para o objeto criado.

Comando ponto I

Para acessar um atributo de um objeto, usa-se a notação ponto:

<nome do objeto>.<nome da variavel>

```
public class Carro {
    String marca;
    String modelo;
    String placa;
    public void acelerar()
    {
    }
}
```

```
...
Carro car1 = new Carro();
Carro car2 = new Carro();
//inicializando car1
car1.marca = "Fiat";
car1.modelo = "2000";
car1.placa = "FRE-6454";
//inicializando car2
car2.marca = "Ford";
car2.modelo = "1995";
car2.placa = "RTY-5675";
...
```

Comando ponto II

Para acessar um método de um objeto, usa-se a notação ponto:

<nome do objeto>.<nome do metodo>

```
public class Carro {
    String marca;
    String modelo;
    String placa;
    public void acelerar()
    { // corpo do metodo
    }
    public void frear()
    { // corpo do metodo
    }
}
```

```
...
Carro car1 = new Carro();
//inicializando car1
car1.marca="Fiat";
car1.modelo="2000";
car1.placa="FRE-6454";
//usando os métodos
car1.acelerar();
car1.frear();
...
```

- Para *mandar mensagens* aos objetos utilizamos o operador ponto, seguido do método que desejamos utilizar;
- Uma **mensagem** em um objeto é a ação de efetuar uma chamada a um método.

```
Pessoa p1;
p1 = new Pessoa();
p1.nome = "Vitor Josue Pereira";
p1.nascimento = "10/07/1966";
p1.gastar( 3200.00 ); // Mensagem sendo passada ao
    objeto p1
```

Um programa orientado a objetos nada mais é do que vários objetos dizendo uns aos outros o que fazer.

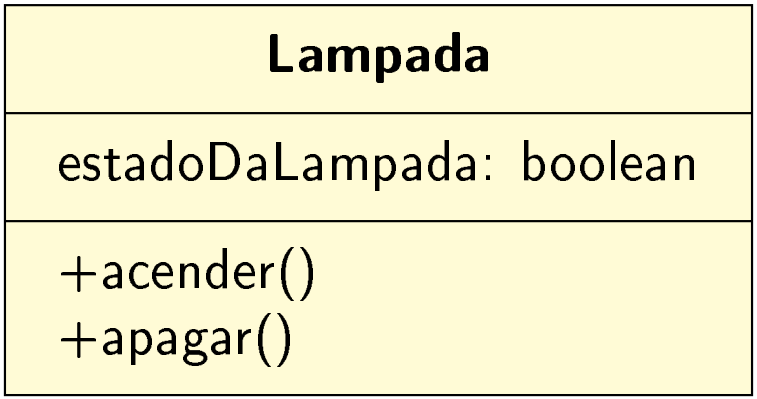
- Quando você quer que um objeto faça alguma coisa, você envia a ele uma "mensagem" informando o que quer fazer, e o objeto faz;
- Se o método for **público**, o objeto terá que executá-lo;
- Se ele precisar de outro objeto para o auxiliar a realizar o "trabalho", ele mesmo vai cuidar de enviar mensagem para esse outro objeto.

O método main()

- O método main() é o ponto de partida para todo aplicativo em Java.
- É nele que são instanciados os primeiros objetos que iniciarão o aplicativo.
- A forma mais usual de se declarar o método main() é mostrada abaixo:

```
public class ClassePrincipal
{
    public static void main (String args [])
    {
        //corpo do método
    }
}
```

Criar um modelo em Java para uma lâmpada. Implemente o modelo, criando dois objetos `Lamp1` e `Lamp2`. Simule a operação `acender` para `Lamp1` e `apagar` para `Lamp2`.



Exercício II

```
1  public class Lampada {
2      boolean estadoDaLampada;
3
4      public void acender( )
5      {
6          estadoDaLampada = true;
7          System.out.println("A acao acender foi executada");
8      }
9      public void apagar( )
10     {
11         estadoDaLampada = false;
12         System.out.println("A acao apagar foi executada");
13     }
14 }
```

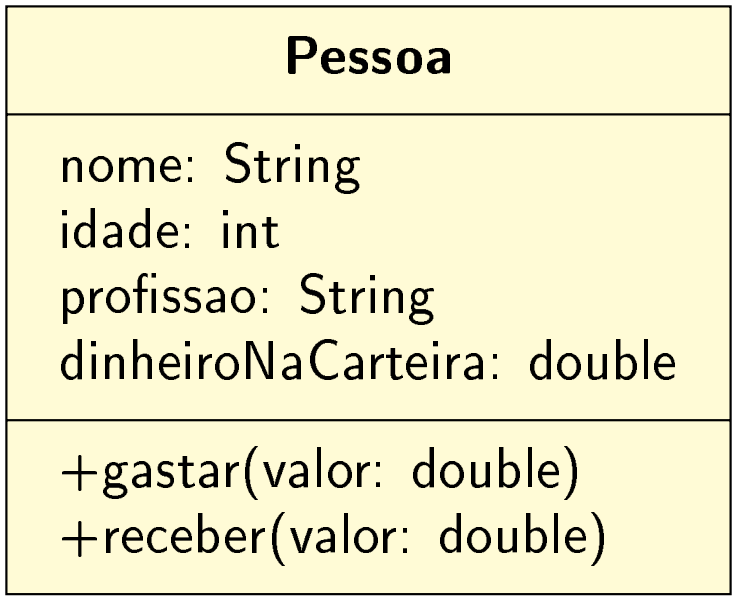
Exercício III

```java
public class GerenciadorDeLampadas {
    public static void main(String args[])  {
        // Declara um objeto Lamp1 da classe Lampada
        Lampada Lamp1;
        // Cria um objeto da classe Lampada
        Lamp1 = new Lampada();
        //Simulando operação sobre objeto Lamp1
        Lamp1.acender();

        // Declara um objeto Lamp2 da classe Lampada
        Lampada Lamp2;
        // Cria um objeto da classe Lampada
        Lamp2 = new Lampada();
        //Simulando operação sobre objeto Lamp2
        Lamp2.apagar();
    }
}
```

Exercício IV

Criar um modelo em Java para uma pessoa. Considere os atributos `nome`, `idade` e `profissão`. A pessoa pode gastar ou receber uma certa quantia de dinheiro (assuma que o dinheiro se encontra na carteira). Implemente o modelo, criando dois objetos `p1` e `p2`. Assuma que o objeto `p1` tem 3.200 na carteira, e o objeto `p2` tem 1.200. Simule operações de gasto e recebimento.

Exercício V


Pessoa
nome: String
idade: int
profissao: String
dinheiroNaCarteira: double
+gastar(valor: double)
+receber(valor: double)

Exercício VI

```java
public class Pessoa
{
    String nome, profissao;
    int idade;
    double dinheiroNaCarteira;
    public void gastar( double valor )
    {
        dinheiroNaCarteira = dinheiroNaCarteira - valor;
    }
    public void receber( double valor )
    {
        dinheiroNaCarteira = dinheiroNaCarteira + valor;
    }
}
```

```java
public class GerenciadorDePessoas
{
    public static void main(String args[])
    {
        // Declara um objeto da classe Pessoa
        Pessoa p1;
        // Cria um objeto da classe Pessoa
        p1 = new Pessoa();
        //Atribuindo valor aos atributos do objeto p1
        p1.nome = "Vitor Pereira";
        p1.idade = 25;
        p1.profissao = "Professor";
        p1.dinheiroNaCarteira = 3200.00;
        System.out.println( "Salário de " + p1.nome + " = " +
        p1.dinheiroNaCarteira );
        // Vitor recebeu 1000 reais
        p1.receber( 1000.00 );
```

```
18        System.out.println( p1.nome + "tem " +
19        p1.dinheiroNaCarteira + " reais");
20        // Vitor gastou 200 reais
21        p1.gastar( 200.00 );
22        System.out.println( p1.nome + "tem agora " +
23        p1.dinheiroNaCarteira + " reais");
24
25        // Declara e cria um outro objeto da classe Pessoa
26        Pessoa p2 = new Pessoa();
27        //Atribuindo valor aos atributos do objeto p2
28        p2.nome = "João Silveira";
29        p2.idade = 30;
30        p2.dinheiroNaCarteira = 1200.00;
31        System.out.println( "Salário de " + p2.nome + " = " +
32        p2.dinheiroNaCarteira );
```

```
33          // João recebeu 400 reais
34          p2.receber( 400.00 );
35          System.out.println( p2.nome + "tem " +
36          p1.dinheiroNaCarteira + " reais");
37          // João gastou 100 reais
38          p2.gastar( 100.00 );
39          System.out.println( p2.nome + "tem agora " +
40          p1.dinheiroNaCarteira + " reais");
41      }
42  }
```

Declaração de variáveis:

```
<tipo> <nomeDaVariável> [= <valorInicial>];
```

```
int x1;
double a, b = 0.4;
float x = -22.7f;
char letra;
String nome = "UFU"
boolean tem;
```

Atribuição de valores:

```
<nomeDaVariável> = <valor>;
```

```
double soma;
soma = 0.6;
obj.nome = "UFU";
```

Desvio condicional simples:

```
if (<condição>)
    <instrução>;


// {}: mais de uma instru
    ção
if (<condição>)
{
    <instruções>;
}
```

Desvio condicional composto:

```
if (<condição>)
{
    <instruções>;
}
else
{
    <instruções>;
}
```

Escolha-caso:

```
switch (expressão)
{
    case <valor1>:
        <comando(s);>
        break;
    case <valor2>:
        <comando(s);>
        break;
        ...
    case <valorN>:
        <comando(s);>
        break;
    default:
        <comando(s);>
}
```

Repetição
Com `while`

```
while (<condição>)
    <comando>;
```

```
while (<condição>)
{
    <comandos>;
}
```

Com `do-while` – teste ao final

```
do
{
    <comandos>;
} while (<condição>);
```

Repetição com o laço `for`

```
for ([tipo] <var = valInicial> ; <condição> ; <
    incremento>)
{
    <comandos>;
}
```

Obs.: passar o tipo quando estiver declarando a variável no comando `for`

Exemplo – variável `i` não declarada antes do `for`:

```
1 for(int i = 1; i <= 10; i++) {
2     <comando 1>;
3     <comando 2>;
4 }
```

Vetores e matrizes I

Vetor (*array*): agrupamento de valores de um mesmo tipo primitivo ou de *objetos de uma mesma classe*.
Em Java, primeiro elemento sempre tem índice 0

- Em vetor de `n` elementos, índices variam de `0` a `n-1`

Exemplo:

0	1	2	3	4
v[0]	v[1]	v[2]	v[3]	v[4]

- Todos os *arrays* são objetos da classe `java.lang.Object` – importada automaticamente, ver Java API adiante;
- Em Java, vetores têm *tamanho fixo* (dado pelo atributo `length`) e **não podem ser redimensionados**;
- Para redimensionar, deve-se criar um novo e fazer cópia.

A utilização de vetores na linguagem Java envolve três etapas:

1. Declarar o vetor;
2. Reservar espaço na memória e definir o tamanho do vetor;
3. Armazenar elementos no vetor.

Para declarar um vetor em Java é preciso acrescentar um par de colchetes antes, ou depois, do nome da variável
Exemplo:

```
int idade[];            // Colchetes depois do
double salario[];       // nome da
String diasDaSemana[];  // variável

double []nota;          // Colchetes antes do
String []nome;          // nome da variável:
char []vet1, vet2, vet3;// Esta notação é útil quando
   queremos declarar mais de um vetor, pois não há
   necessidade de repeti-los.
```

É preciso definir o tamanho do vetor – isto é, a quantidade total de elementos que poderá armazenar;
Em seguida é necessário reservar espaço na memória para armazenar os elementos. Isto é feito pelo operador `new`.

```
// Sintaxe 1
int idade[];
idade = new int[10];
double salario[];
salario = new double[6];

// Sintaxe 2
double []nota = new double [125];
String nome[] = new String [70];
```

A cópia de vetores é possível através de um método da classe `System`, denominado `arraycopy()`:

```
int v1[] = {1,2,3,4,5};
v1[5] = 10; // erro: índice fora dos limites

int v2[] = new int[10];
System.arraycopy(v1, 0, v2, 0, v1.length);
v2[5] = 10; // ok, os índices de v2 vão de 0 a 9
```

Sintaxe: `arraycopy(v1,i1,v2,i2,n)`: Copia n elementos do vetor `v1` – a partir do elemento de índice $i1$ – para o vetor `v2`, a partir do índice $i2$.

Conforme já adiantado no exemplo anterior, há um atalho que resume os três passos vistos anteriormente (declaração, reserva de espaço, atribuição de valor).
Outro exemplo deste atalho:

```
// 10 primeiros elementos da sequência de Fibonacci
long fibonacci[] = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
```

Também é possível armazenar **objetos** em vetores:

```
Pessoa [] clientes = new Pessoa[3];
clientes[0] = new Pessoa(); // Necessário neste caso
clientes[0].nome = "João";
clientes[0].idade = 33;
clientes[0].profissao = "gerente";
clientes[1] = new Pessoa();
clientes[1].nome = "Pedro";
clientes[1].idade = 25;
clientes[1].profissao = "caixa";
clientes[2] = new Pessoa();
clientes[2].nome = "Maria";
clientes[2].idade = 20;
clientes[2].profissao = "estudante";
```

Percorrendo *arrays*: podemos fazer `for` especificando os índices – como fazemos em C.
Porém, se o **vetor todo será percorrido**, podemos usar o ***enhanced for***:

```
public void somaVetor()
{
    int[] vetor = {87, 68, 94, 100, 68, 39, 10};
    int total = 0;
    for (int valor: vetor) // enhanced for
        total = total + valor;
        System.out.println("Total = " + total);
}
```

Vetores multidimensionais:

- Vetor bidimensional: **matriz**.
  - Ex.: `double[][] matriz = new double[5][10];`
- É possível construir vetores multidimensionais **não retangulares**:

```
// matriz de inteiros, 2 x 3 (2 linhas e 3 colunas)
int v[][] = new int[2][];
v[0] = new int[3];
v[1] = new int[3];
// vetor bidimensional não-retangular.
int vet[][] = new int[3][];
vet[0] = new int[2];
vet[1] = new int[4];
vet[2] = new int[3];
```

Em Java, é possível criar métodos que recebem um número não especificado de parâmetros – **usa-se um tipo seguido por reticências**:

- Método recebe número variável de parâmetros desse tipo;
- Reticências podem ocorrer apenas uma vez;
- No corpo do método, a lista variável é tratada como um vetor.

```
public class Media {
    public double media(double... valor) {
        double soma = 0.0;
        for (int i=0; i<valor.length; i++)
            soma = soma + valor[i];
        return soma/valor.length;
    }
}
```

Java API (*Applications Programming Interface*): conjunto de classes pré-definidas do Java;
API está organizadas em pastas (pacotes)

Exemplos:

Pacote	Descrição
`java.lang`	Classes muito comuns (este pacote é **importado automaticamente** pelo compilador `javac` em todos os programas Java).
`java.io`	Classes que permitem entrada e saída em arquivos.
`java.math`	Classes para executar aritmética de precisão arbitrária
`javax.swing`	Classes de componentes GUI Swing para criação e manipulação de interfaces gráficas do usuário
`java.util`	Utilitários como: manipulações de data e hora, processamento de números aleatórios, armazenamento e processamento de grandes volumes de dados, quebras de *strings* em unidades léxicas etc.

A classe `java.lang.Math`, por exemplo, contém valores de constantes, como `Math.PI` e `Math.E`, e várias funções matemáticas. Alguns métodos:

Método	Descrição
`Math.abs(x)`	Valor absoluto de `x`.
`Math.ceil(x)`	Teto de `x` (menor inteiro maior ou igual a `x`).
`Math.cos(x)`	Cosseno de `x` (`x` dado em radianos).
`Math.exp(x)`	Exponencial de `x` (e^x).
`Math.floor(x)`	Piso de `x` (maior inteiro menor ou igual a `x`).

Métodos estáticos

Chamados a partir da classe (e não de um objeto específico)

A classe `String` opera sobre *cadeias de caracteres*.

Método	Descrição
`ob.concat(s)`	Concatena objeto `ob` ao `String s`.
`ob.replace(x,y)`	Substitui, no objeto `ob`, todas as ocorrências do caractere `x` pelo caractere `y`.
`ob.substring(i,j)`	Constrói novo String para `ob`, com caracteres do índice i ao índice $j-1$.

Métodos não são estáticos

Chamados a partir de objetos da classe `String`.

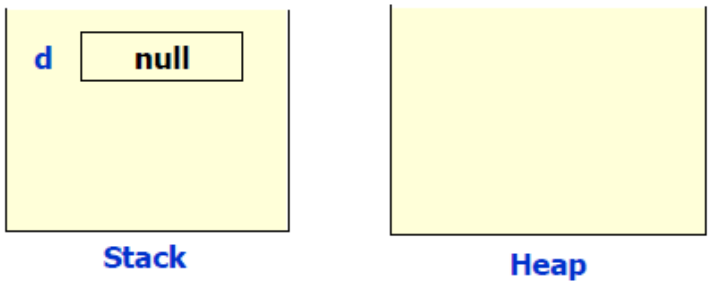
Variáveis de tipos primitivos ou referências são criados em uma área de memória conhecida como **Stack** (pilha);
Por sua vez, objetos são criados em área de memória conhecida como **Heap** (monte). Uma instrução `new Xxxx()`:

- Aloca memória para a variável de referência ao objeto na pilha e inicia-a com valor **null**;
- Executa construtor, aloca memória na *heap* para o objeto e inicia seus campos (atributos);
- Atribui endereço do objeto na *heap* à variável de referência do objeto na pilha.

Armazenamento em memória II

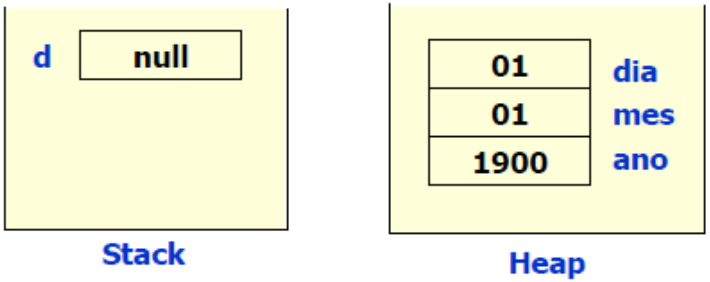
1

```
MinhaData d = new MinhaData (17, 3, 2009);
```



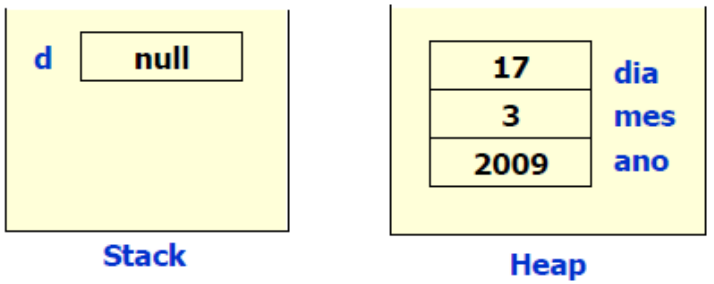
2

```
MinhaData d = new MinhaData (17, 3, 2009);
```



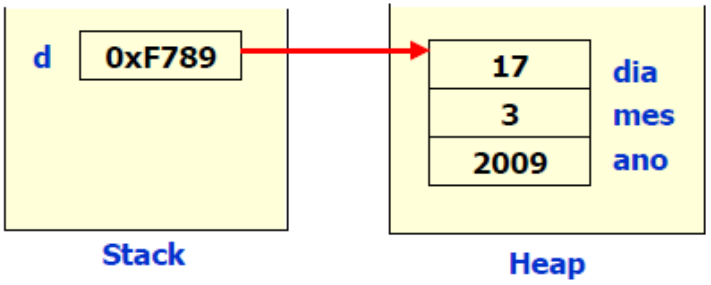
3

```
MinhaData d = new MinhaData (17, 3, 2009);
```



4

```
MinhaData d = new MinhaData (17, 3, 2009);
```

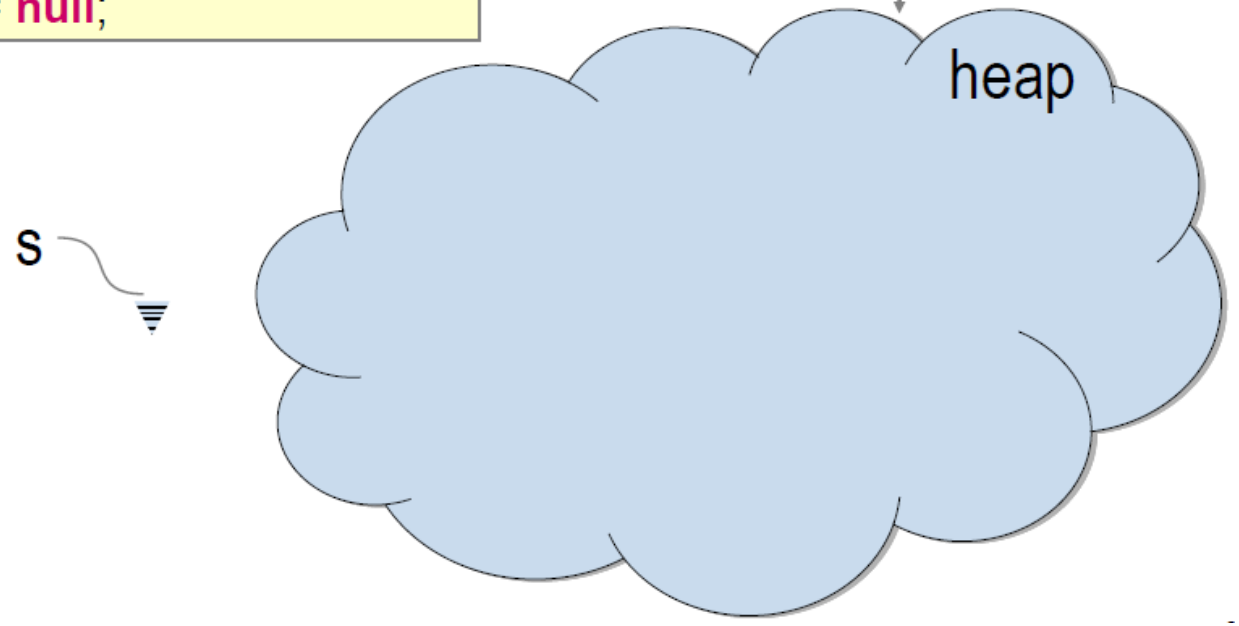


- Periodicamente, a JVM realiza uma **coleta de lixo**:
  - Retorna partes de memória não usadas;
  - Objetos não referenciados;
  - Não existem então primitivas para alocação e desalocação de memória (como `calloc`, `free`, etc).

Coleta de lixo II

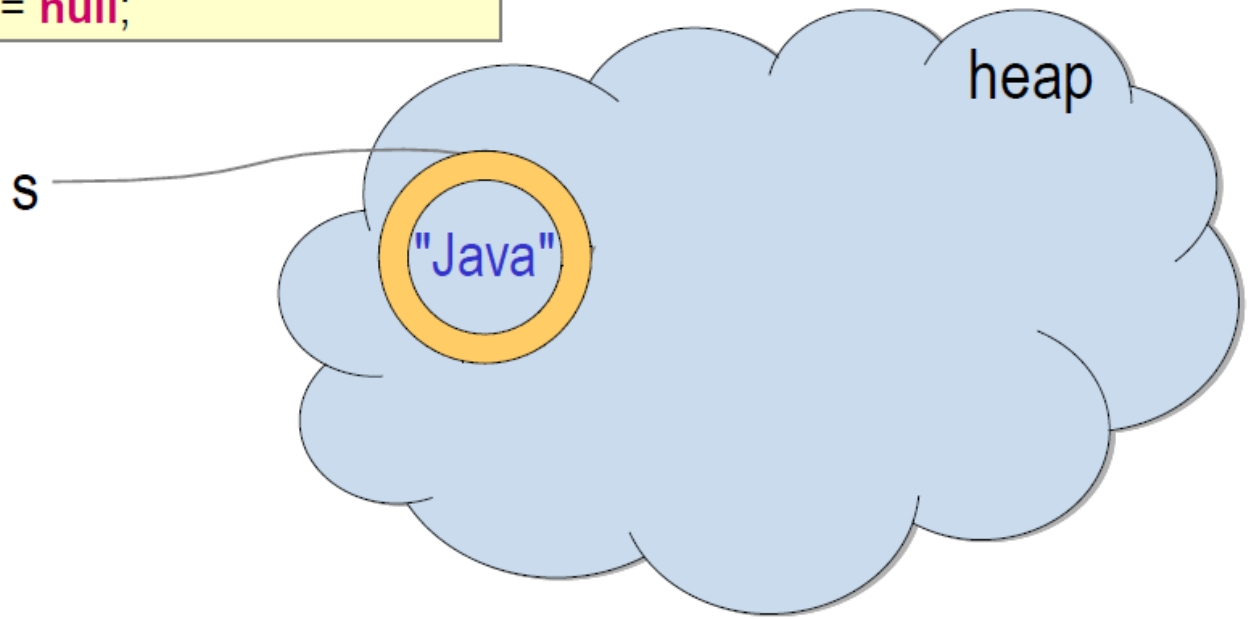
```
String s;
s = "Java";
s = null;
```

Área onde os objetos são criados e mantidos pela JVM



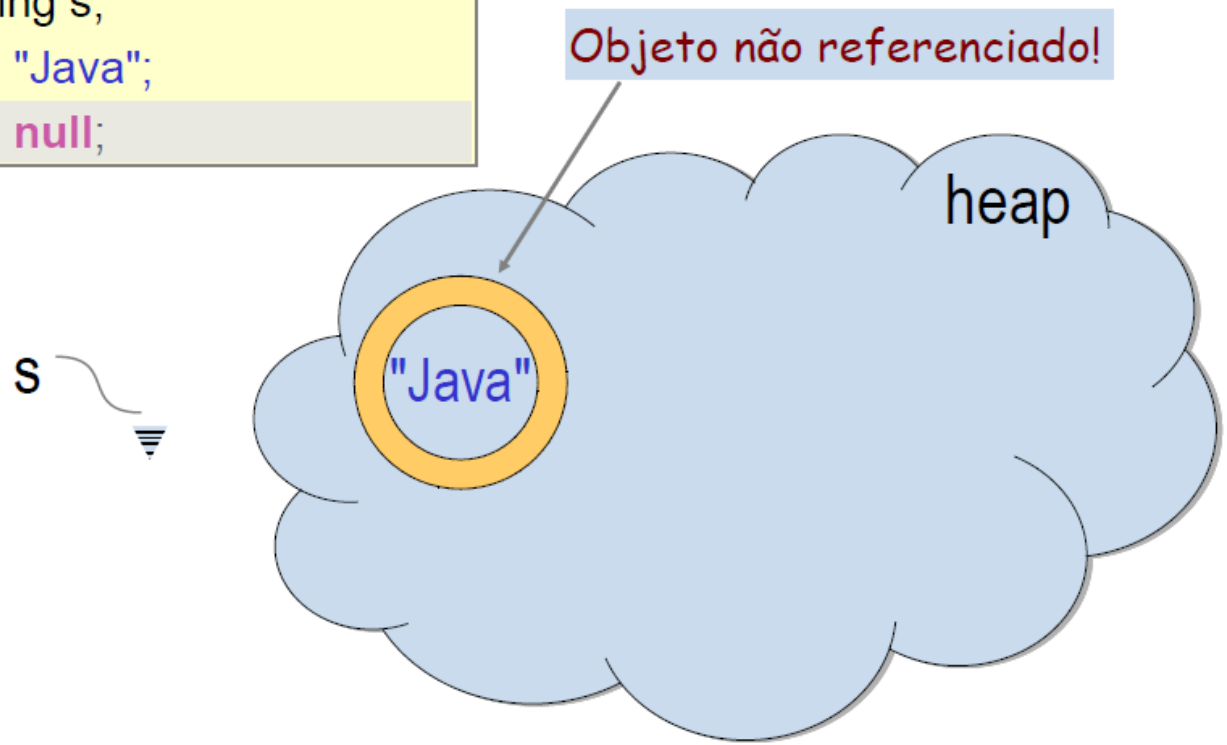
Coleta de lixo III

```
String s;
s = "Java";
s = null;
```



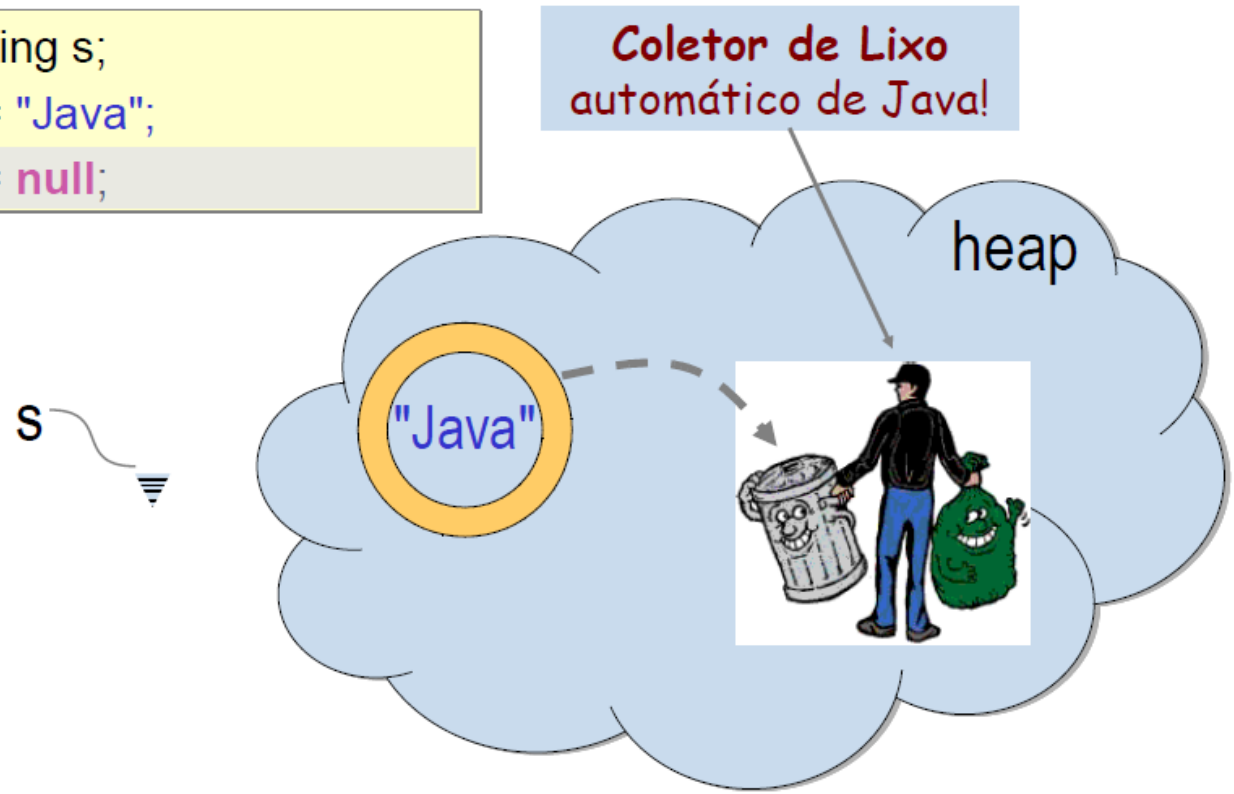
Coleta de lixo IV

```
String s;
s = "Java";
s = null;
```



Coleta de lixo V

```
String s;
s = "Java";
s = null;
```



Encapsulamento e modificadores de acesso I

A linguagem Java oferece mecanismos de controle de acessibilidade (visibilidade):

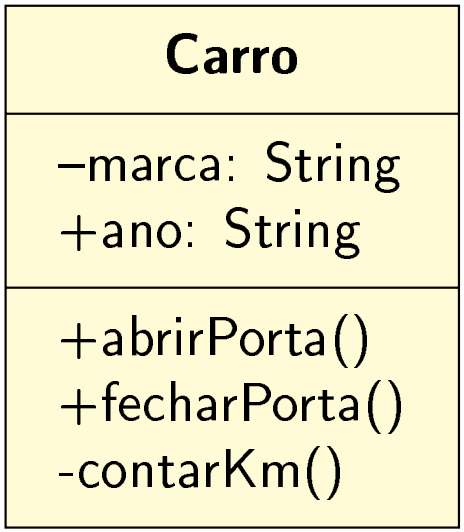
Encapsulamento

Encapsulamento é a capacidade de controlar o acesso a classes, atributos e métodos

- O encapsulamento é implementado através dos **modificadores de acesso**;
  - São palavras reservadas que permitem definir o encapsulamento de classes, atributos e métodos;
- **Modificadores de acesso**: `public`, `private`, `protected`.
- Quando se omite, o acesso é do tipo *package-only*.
- Classes: apenas `public` – ou omitido (*package-only*) – é permitido.

- *Package-only*
  - Caso padrão (quando modificador de acesso é omitido);
  - Permite acesso a partir das classes do mesmo pacote – ou classe de outro pacote, desde que seja *subclasse* (herdeira) da classe em questão (conceito de herança).
- `protected`
  - Permite acesso a partir de uma classe que é herdeira de outra.
- `public`
  - Permite acesso irrestrito a partir de qualquer classe (mesmo que estejam em outros arquivos);
  - Único que pode ser usado em classes.
- `private`
  - Permite acesso apenas por objetos da própria classe. O elemento é visível apenas dentro da classe onde está definido.

- +: `public` – visível em qualquer classe;
- –: `private` – visível somente dentro da classe.
- #: `protected` – visibilidade associada à herança




```
1  public class Carro
2  {
3      private String marca;
4      public String ano;
5      public void abrirPorta()
6      {
7          //corpo do método
8      }
9      public void fecharPorta()
10     {
11         //corpo do método
12     }
13     private void contarKm()
14     { /*corpo do método*/ }
15 }
```

Encapsulamento e diagrama de classes II

```java
public class Carro
{
    private String marca;
    public String ano;
    public void abrirPorta()
    {
        //corpo do método
    }
    public void fecharPorta()
    {
        //corpo do método
    }
    private void contarKm()
    { /*corpo do método*/ }
}
```

```java
public class UsaCarro
{
    public static void main(String
          args[])
    {
        Carro car1 = new Carro();
        car1.ano = "2000";
        car1.marca = "Fiat";

        car1.fecharPorta();
        car1.contarKm();
    }
}
```

Erros de semântica: linhas 7 e 10.

- Um método `public` pode ser invocado (chamado) dentro da própria classe, ou a partir de qualquer outra classe;
- Um método `private` é acessível apenas dentro da classe a que pertence.

- Atributos públicos podem ser acessados e modificados a partir de qualquer classe;
- A menos que haja razões plausíveis, os atributos de uma classe devem ser definidos como `private`;
- Tentar acessar um componente privado de fora da classe resulta em **erro de compilação**.

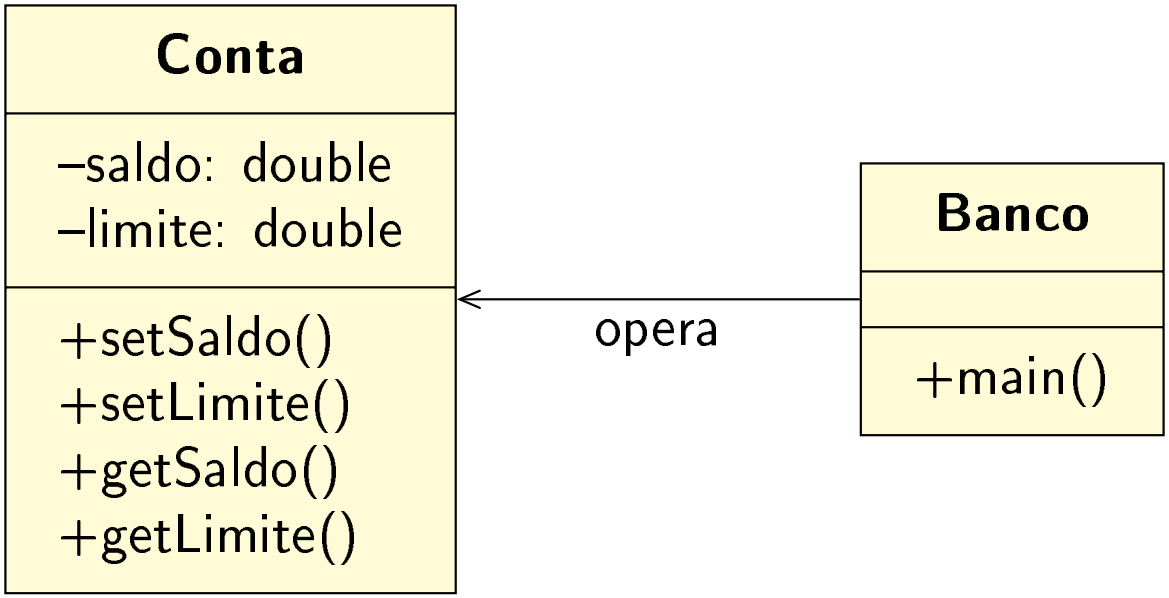
Mas então como acessar atributos, ao menos para consulta (leitura)?

- Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar *dois métodos*;
- Os dois métodos são definidos na própria classe onde o atributo se encontra;
- Um dos métodos retorna o valor da variável
- Outro método muda o seu valor;
- Padronizou-se nomear esses métodos colocando a palavra *get* ou *set* antes do nome do atributo.

- Com atributos sendo `private`, é frequente usar **métodos acessores/modificadores** (*getters*/*setters*) para manipular atributos;
- Porém, devemos ter cuidado para não quebrar o encapsulamento:

Se uma classe chama `objeto.getAtrib()`, manipula o valor do atributo e depois chama `objeto.setAtrib()`, o atributo é essencialmente público. De certa forma, estamos quebrando o encapsulamento!

Construa um programa para simulação de um banco que possui apenas uma `conta`, com os atributos privados `saldo` e `limite`. Utilize métodos *getters* e *setters* para manipular os valores dos atributos e visualizá-los. Uma entidade `banco` é responsável pela criação da conta e sua operação.



Classe Conta.java (versão 0)

```
1  public class Conta {
2      private double limite;
3      private double saldo;
4      public double getSaldo() {
5          return saldo;
6      }
7      public void setSaldo(double x) {
8          saldo = x;
9      }
10     public double getLimite() {
11         return limite;
12     }
13     public void setLimite(double y) {
14         limite = y;
15     }
16 }
```

Questão

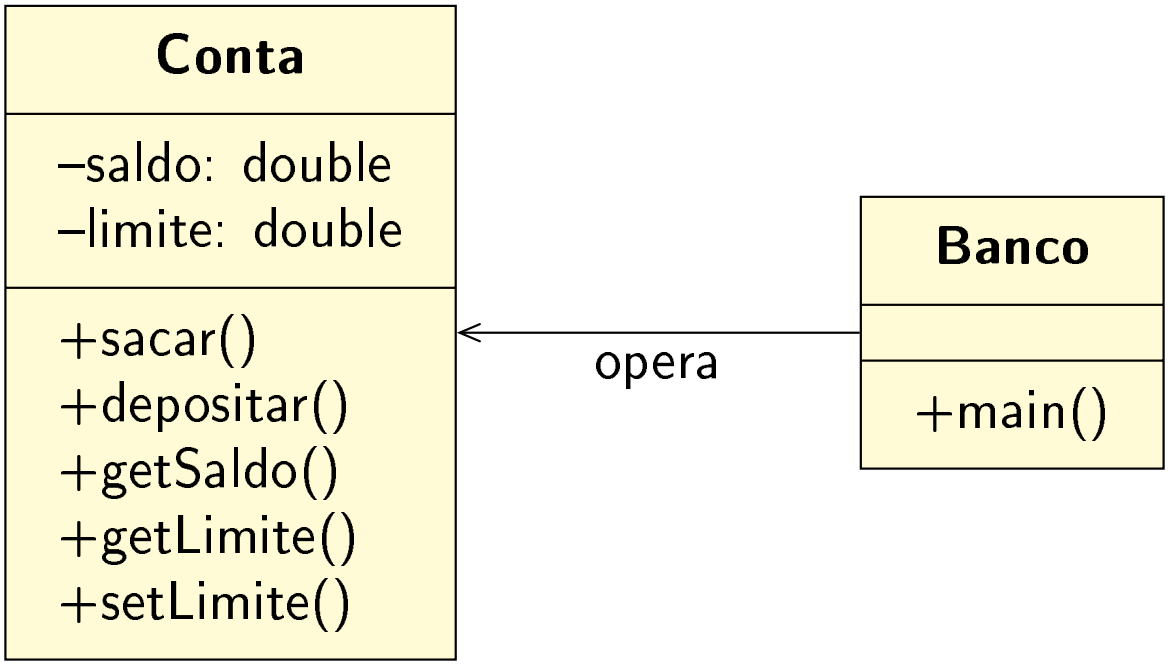
Esta classe permite alterar seus atributos como se fossem públicos!

Classe Banco.java (versão 0)

```
1  public class Banco
2  {
3      public static void main ( String args[] )
4      {
5          Conta c1 = new Conta();
6          c1.setSaldo(  1000 );
7          c1.setLimite( 1000 );
8          double saldoAtual = c1.getSaldo();
9          System.out.println( "Saldo atual é " + saldoAtual );
10         double limiteConta = c1.getLimite();
11         System.out.println( "Limite é " + limiteConta );
12     }
13 }
```

Exercício – simulação de um banco (v1)

Construa um programa para simulação de um banco que possui apenas uma `conta`, com os atributos privados `saldo` e `limite`. Uma entidade `banco` é responsável pela criação da conta e sua operação. Ela pode sacar e depositar dinheiro da conta, visualizar seu saldo atual, assim como verificar e atualizar o limite.



Classe Conta.java (versão 1)

```
1  public class Conta {
2      private double saldo;
3      private double limite;
4      public void depositar( double
           x )
5      {
6          saldo = saldo + x;
7      }
8      public void sacar( double x )
9      {
10         saldo = saldo - x;
11     }
12     public double getSaldo()
13     {
14         return saldo;
15     }
16     public void setLimite( double
           x )
17     {
18         limite = x;
19     }
20
21     public double getLimite()
22     {
23         return limite;
24     }
25 }
```

Já colocamos nos métodos **regras de negócio**, que representam a lógica de negócio da sua aplicação.

Classe Banco.java (versão 1) I

```
1  public class Banco
2  {
3      public static void main ( String args[] )
4      {
5          Conta c1 = new Conta();
6          c1.setLimite( 300 );
7          c1.depositar( 500 );
8          c1.sacar( 200 );
9          System.out.println( "O saldo é " + c1.getSaldo() );
10     }
11 }
```

E se sacarmos mais que o disponível?

Reescrevemos o método `sacar()`:

```
8     public void sacar(double x) {
9        if ( saldo + limite >= x )
10          saldo = saldo - x;
11    }
```

Métodos permitem controlar os valores / atributos, evitando que qualquer objeto altere seu conteúdo sem observar regras.

Métodos construtores são utilizados para realizar *toda a inicialização necessária* a uma nova instância da classe;

- Diferente de outros métodos, um método construtor não pode ser chamado diretamente:
  Um construtor é invocado pelo operador `new` quando um novo objeto é criado;
- Determina como um objeto é inicializado quando ele é criado;
- **Vantagens**: não precisa criar métodos *get*/*set* para cada um dos atributos privados da classe (reforçando o encapsulamento), tampouco enviar mensagens de atribuição de valor após a criação do objeto.

A declaração de um método construtor é semelhante a qualquer outro método, com as seguintes particularidades:

- O nome do construtor **deve ser o mesmo da classe**;
- Um construtor não possui um tipo de retorno – sempre `void`.

Classe Veiculo.java I

```
1  class Veiculo
2  {
3      private String marca;
4      private String placa;
5      private int kilometragem;
6      public Veiculo( String m, String p, int k )
7      {
8          marca = m;
9          placa = p;
10         kilometragem = k;
11     }
12     public String getPlaca()
13     {
14         return placa;
15     }
16     public String getMarca()
17     {
18         return marca;
19     }
```

Classe Veiculo.java II

```
20      public int getKilometragem()
21      {
22          return kilometragem;
23      }
24      public void setKilometragem(int k)
25      {
26          kilometragem = k;
27      }
28  }
```

Classe AcessaCarro.java

```
1  class AcessaCarro
2  {
3      public static void main(String args[])
4      {
5          Veiculo meuCarro = new Veiculo("Escort","XYZ-3456",60000);
6          String marca;
7          int kilometragem;
8          marca = meuCarro.getMarca();
9          System.out.println( marca );
10         kilometragem = meuCarro.getKilometragem();
11         System.out.println( kilometragem );
12         meuCarro.setKilometragem( 100000 );
13         System.out.println( kilometragem );
14     }
15 }
```

Referências

- HORSTMANN, Cay S.; CORNELL, Gary. *Core Java 2: Vol.1 – Fundamentos*, Alta Books, SUN Mircosystems Press, 7a. Edição, 2005.
- DEITEL, H. M.; DEITEL, P. J. *JAVA – Como Programar*, Pearson Prentice-Hall, 6a. Edição, 2005.
- https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

UML – Linguagem de modelagem unificada I

- **UML** (*Unified Modeling Language*) – linguagem visual muito utilizada nas etapas de **análise** e **projeto** de sistemas computacionais no paradigma de orientação a objetos
- A **UML** se tornou a linguagem padrão de projeto de software, adotada internacionalmente pela indústria de Engenharia de Software.

- UML não é uma linguagem de programação: é uma **linguagem de modelagem** utilizada para representar o sistema de software sob os seguintes parâmetros:
  - Requisitos
  - Comportamento
  - Estrutura lógica
  - Dinâmica de processos
  - Comunicação/interface com os usuários

- O objetivo da UML é fornecer múltiplas visões do sistema que se deseja modelar, representadas pelos **diagramas UML**;
- Cada diagrama analisa o sistema sob um determinado aspecto, sendo possível ter enfoques mais amplos (externos) do sistema, bem como mais específicos (internos).

- Diagrama de casos e usos
- **Diagrama de classes**
- Diagrama de objetos
- Diagrama de sequência
- Diagrama de colaboração
- Diagrama de estado
- Diagrama de atividades

Softwares para UML

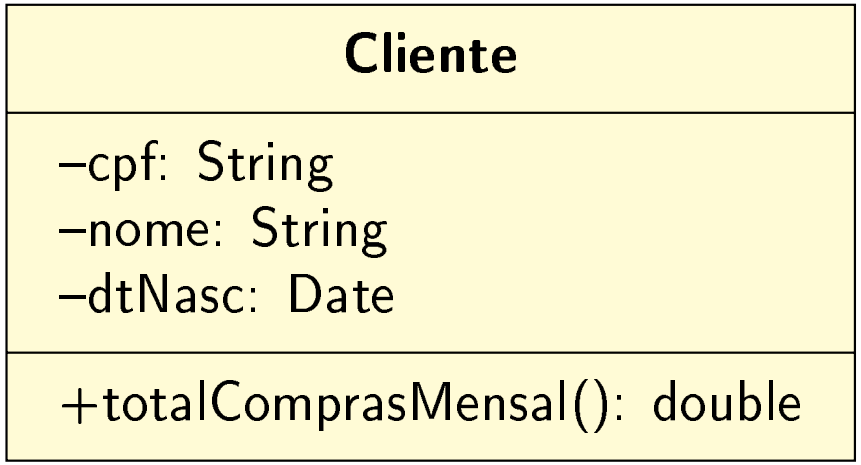
Algumas ferramentas para criação de diagramas UML:

- Rational Rose – a primeira ferramenta case a fornecer suporte UML
- Yed
- StarUML – Ferramenta OpenSource
- Dia
- Argo UML
- Microsoft Visio
- Enterprise Architect

- **Mais utilizado** da UML
- Objetivos:
  - Ilustrar as classes principais do sistema;
  - Ilustrar o relacionamento entre os objetos.
- Apresentação da estrutura lógica: classes, relacionamentos.

Nos diagramas UML, cada classe é dada por um **retângulo** dividido em três partes:

1. O **nome** da classe;
2. Seus **atributos**;
3. Seus **métodos**.



Atributos no diagrama de classe:
visibilidade nome: tipo = valor inicial {propriedades}

- **Visibilidade**: + para `public`, − para `private`, # para `protected`;
- **Tipo** do atributo: Ex: `int`, `double`, `String`, `Date`;
- **Valor inicial** definido no momento da criação do objeto;
- **Propriedades**. Ex.: {`readonly`, `ordered`}

Métodos no diagrama de classe:
visibilidade nome (par1: tipo1, par2: tipo2, ...): tipo

- **Visibilidade**: + para `public`, − para `private`, # para `protected`;
- **(par1: tipo1, par2:tipo2, ...)**: Se método contém parâmetros formais (argumentos), o nome e o tipo de cada 1. Ex: `(nome: String, endereco: String, codigo: int)`;
  - Se método não contém parâmetros, manter par de parênteses, vazio.
- **tipo**: tipo de retorno do método. Ex.: `void`, se não retorna nada; `int`, `Cliente` (nome de uma classe), etc.

Mais detalhes sobre UML:
http://www.uml.org/what-is-uml.htm

- Como os objetos das diferentes classes se relacionam?
- Como **diferentes classes** se relacionam?
- Vamos identificar as principais formas de conexão entre classes:
  - E representar esses relacionamentos no **diagrama de classes**;
  - Relações fornecem um **caminho** para a comunicação entre os objetos.

Tipos mais comuns:

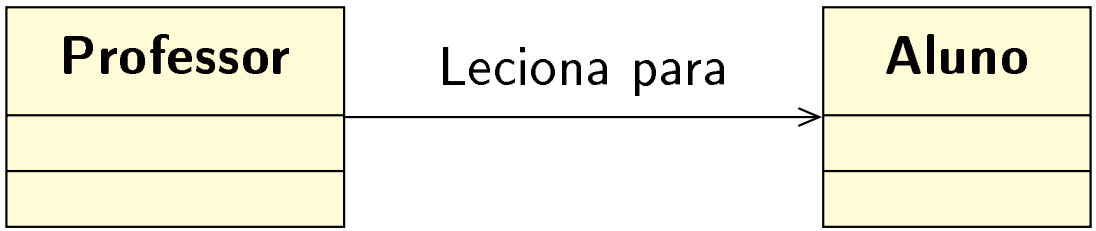
- Entre **objetos de diferentes classes**:
  - **Associação** – "usa";
  - **Agregação** – "é parte de";
  - **Composição** – "é parte essencial de".
- Entre **classes**:
  - **Generalização** – "É um"

Relacionamentos são caracterizados por:

- **Nome**: descrição dada ao relacionamento (*faz*, *tem*, *possui*,...);
  - É usualmente um verbo.
- **Navegabilidade**: indicada por uma seta no fim do relacionamento;
  - Uni- ou bidirecional.
- **Multiplicidade**: 0..1, 0..*, 1, 1..*, 2, 3..7
- **Tipos de relacionamentos**: associação simples, agregação, composição, generalização.

Para facilitar seu entendimento, uma associação pode ser nomeada.

- O nome é representado como um "rótulo" colocado ao longo da linha de associação;
- Um nome de associação é usualmente um verbo ou uma frase verbal.



Multiplicidade refere-se ao número de instâncias de uma classe relacionada com uma instância de outra classe.
Para cada associação, há duas decisões a fazer, uma para cada lado da associação.
Por exemplo, na conexão entre Professor e Aluno:

- Para cada instância de Professor, podem ocorrer muitos (zero ou mais) Alunos;
- Por outro lado, para cada instância de Aluno, pode ocorrer exatamente um Professor (pensando em um curso isolado).

Multiplicidade II

Tipos de multiplicidade:

Muitos (equivale a *0 ou mais*)	*
Zero ou mais	0..*
Exatamente um	1
Um ou mais	1..*
Zero ou um	0..1
Faixa especificada (ex.: entre 2 e 7)	2..7
m ou *n*	*m*, *n*

Exemplos:

- Um cliente pode ter apenas um nome;
- Uma turma deve ter no mínimo cinco e no máximo 100 alunos;
- Uma mesa de restaurante pode ter vários ou nenhum pedido;
- Uma cotação pode ter no mínimo um item.

É a forma **mais fraca** de relacionamento entre classes:

- As classes que participam desse relacionamento são independentes;
- Pode envolver duas ou mais classes.

Representa relacionamentos "usa":

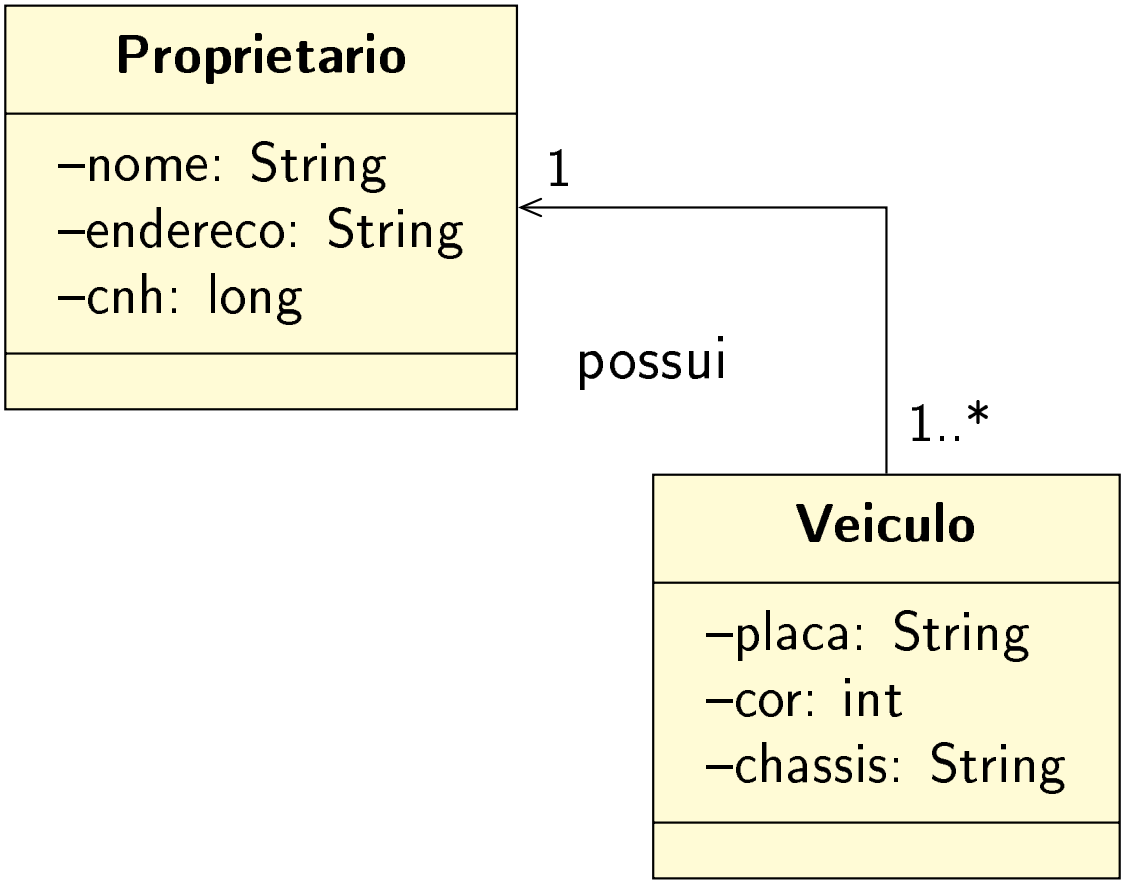
- Ex.: uma pessoa "usa um" carro
- **Na implementação**: um objeto A usa outro objeto B *chamando um método público de B*.

No diagrama de classes, as associações:

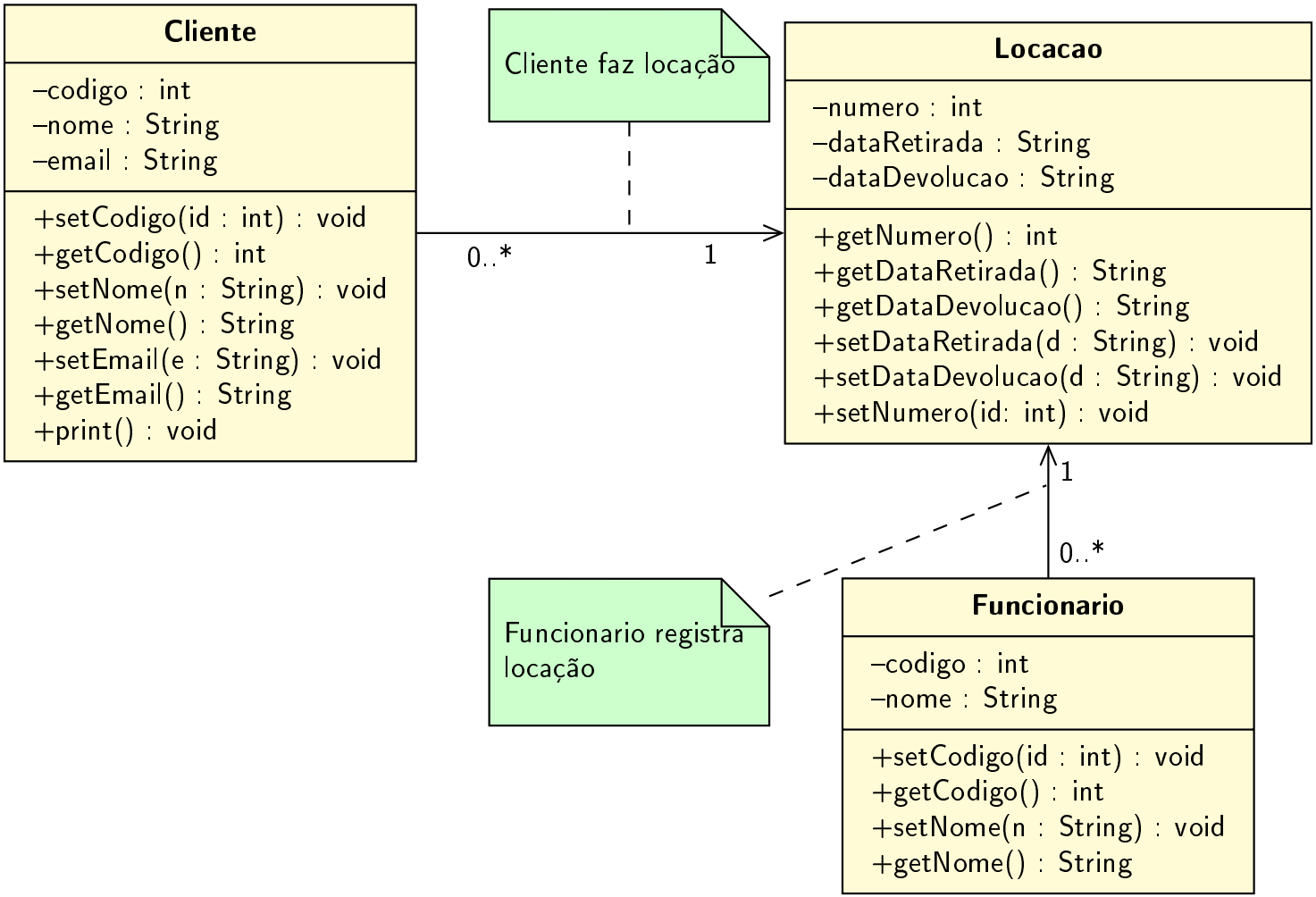
- São representadas como linhas conectando as classes participantes do relacionamento;
- Podem ter um nome identificando a associação;
- Podem ter uma seta junto ao nome indicando que a associação somente pode ser utilizada em uma única direção.

Exemplos:

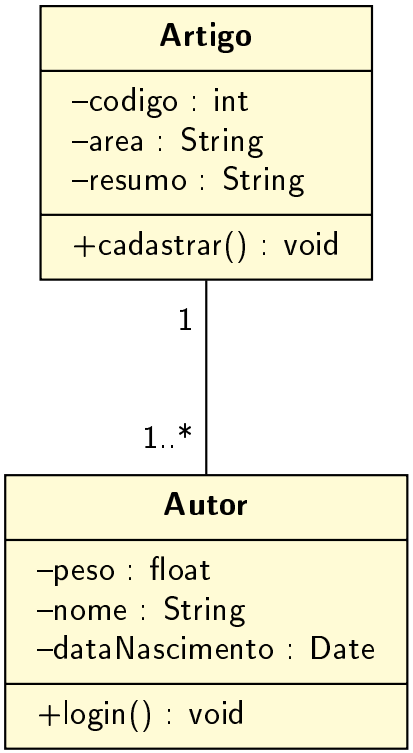
Associação simples IV



Associação simples V



Associação simples VI



Imagine um sistema de avaliação de artigos acadêmicos:

- Temos uma relação autor/artigo;
- Note que a classe Autor não compartilha atributos da classe Artigo e vice-versa;
- Nesse caso, não existe a relação *todo-parte*.

Essas duas formas de associação representam relacionamentos do tipo **"tem um"**

- Relacionamento **todo-parte** $\Rightarrow$ Uma classe é formada por, ou contém objetos de outras classes

Exemplos:

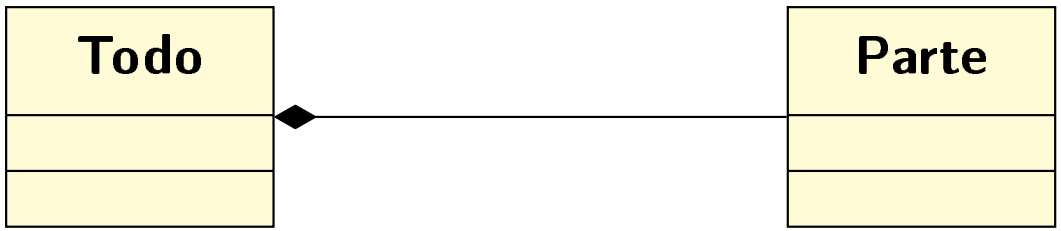
- Um carro contém portas;
- Uma árvore é composta de folhas, tronco, raízes;
- Um computador é composto de CPU, teclado, mouse, monitor, ...

A **agregação** é uma forma mais fraca de **composição**.

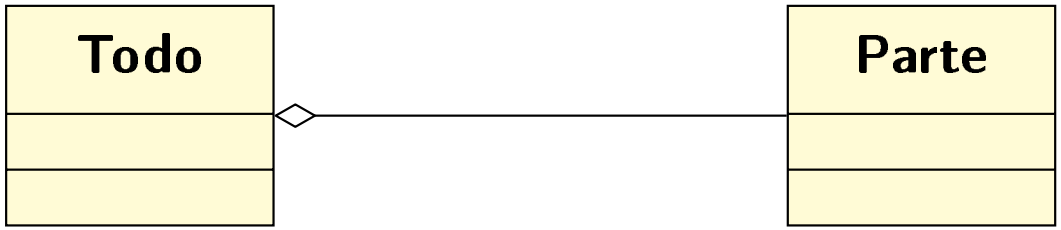
- **Composição**: relacionamento todo-parte em que as **partes** não podem existir independentes do **todo**:
  - Se o *todo* é destruído as *partes* são destruídas também;
  - Uma *parte* pode ser de um único *todo* por vez.
- **Agregação**: relacionamento todo-parte que não satisfaz um ou mais desses critérios:
  - A destruição do objeto *todo* não implica a destruição do objeto *parte*;
  - Um objeto pode ser *parte* componente de vários outros objetos.

Representação:

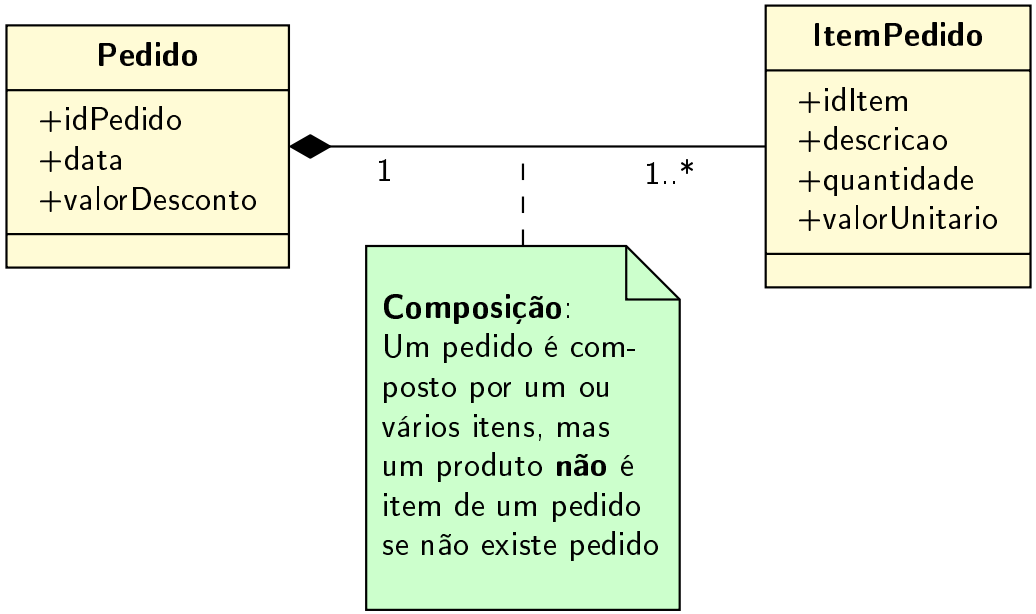
- **Composição**: associação representada com um losango **sólido** do lado *todo*.
  - O lado *todo* deve sempre ter multiplicidade 1.



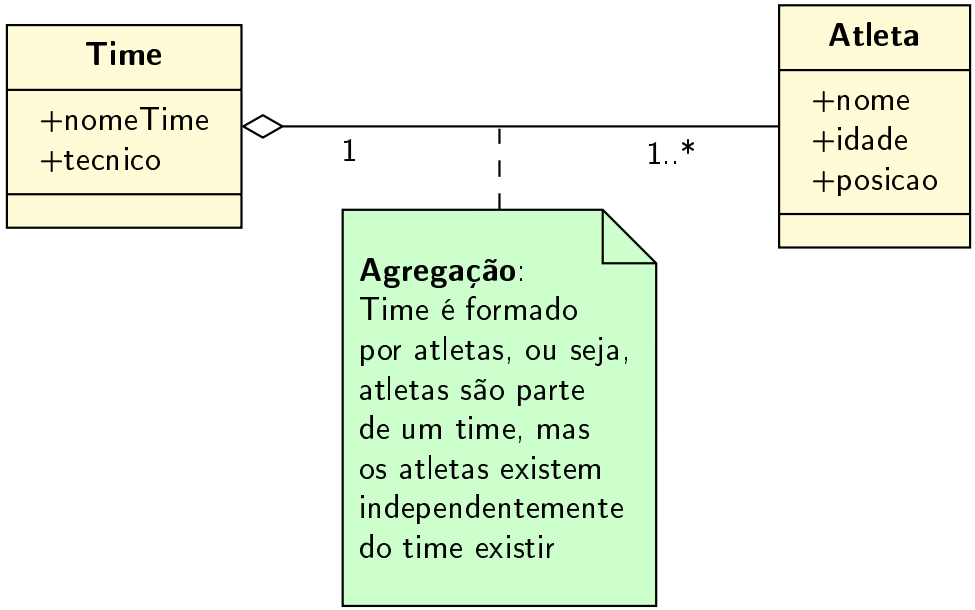
- **Agregação**: associação representada com um losango **sem preenchimento** do lado *todo*.

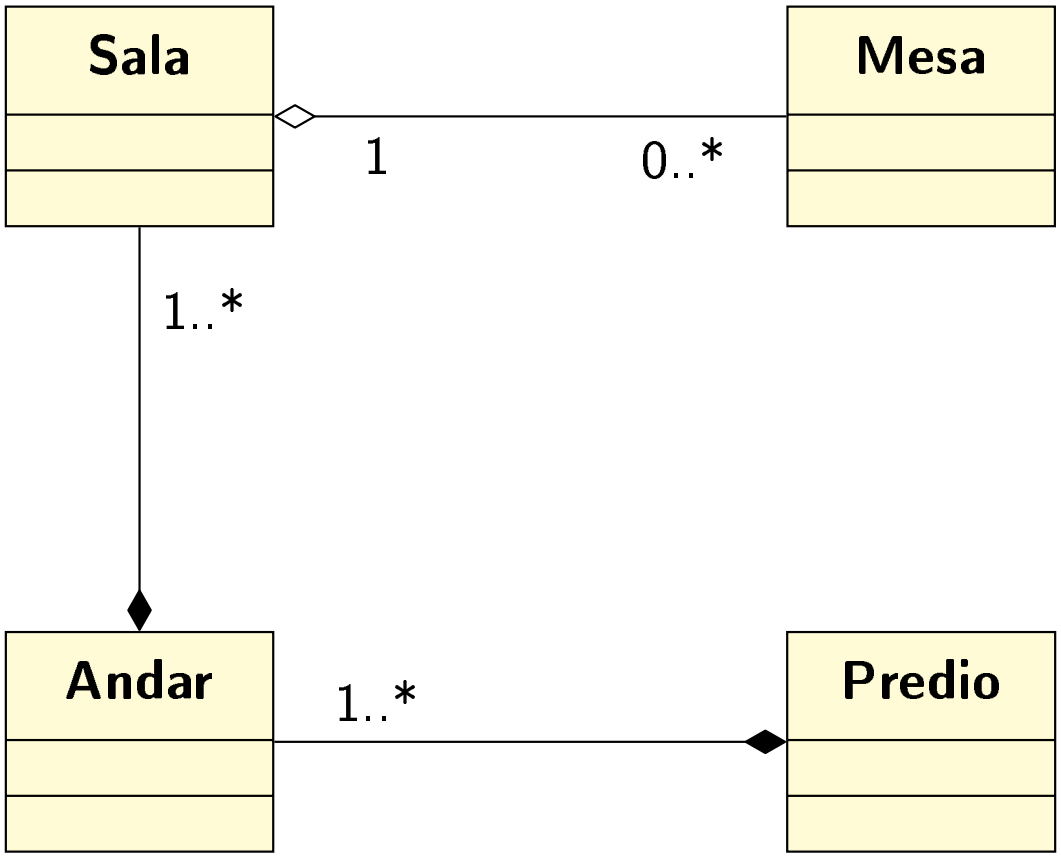


Exemplos:

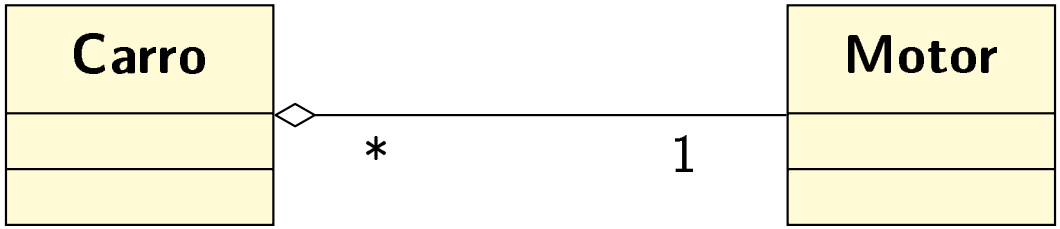

Pedido
+idPedido
+data
+valorDesconto
1
1..*
ItemPedido
+idItem
+descricao
+quantidade
+valorUnitario
Composição:
Um pedido é composto por um ou vários itens, mas um produto não é item de um pedido se não existe pedido

Agregação e Composição V

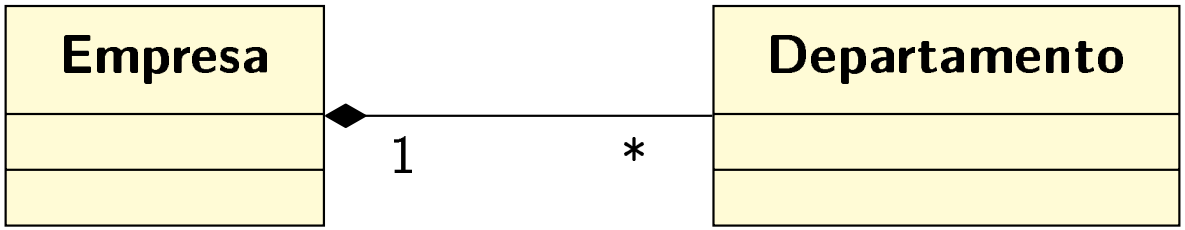



Sala
Mesa
1
0..*
1..*
Andar
1..*
Predio

Agregação:



Composição:

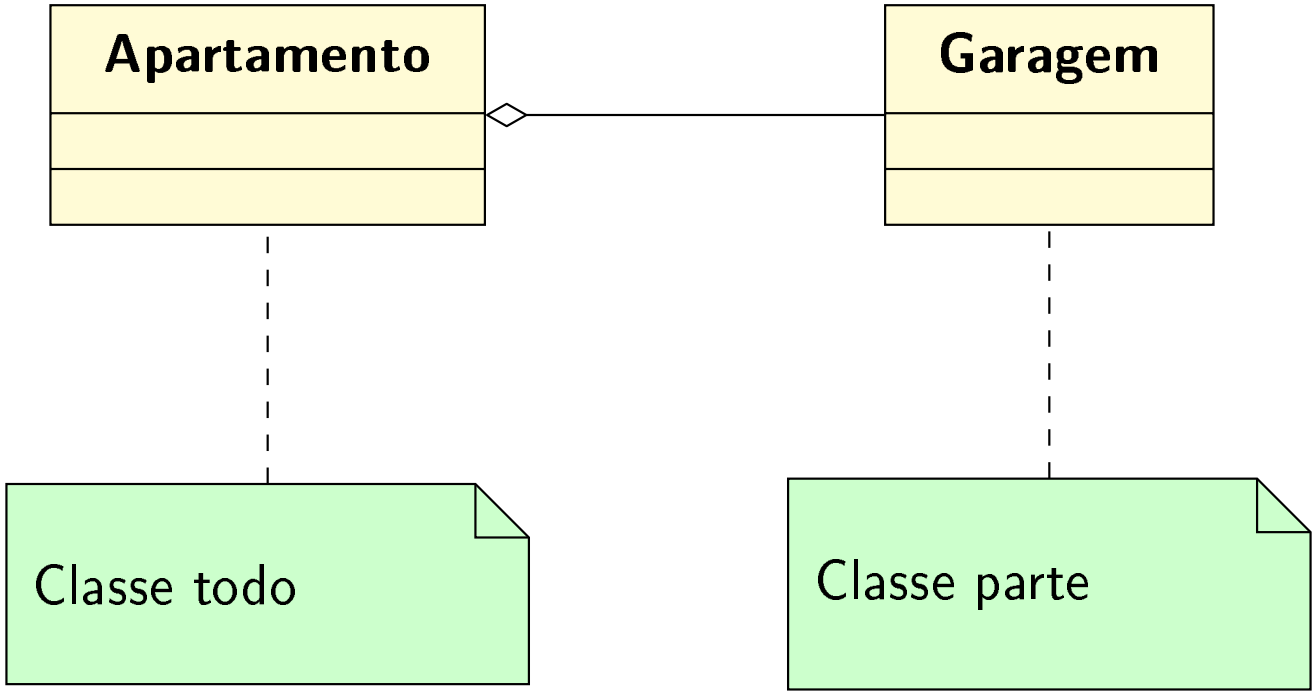


Classe todo:
É a classe resultante da agregação ou composição

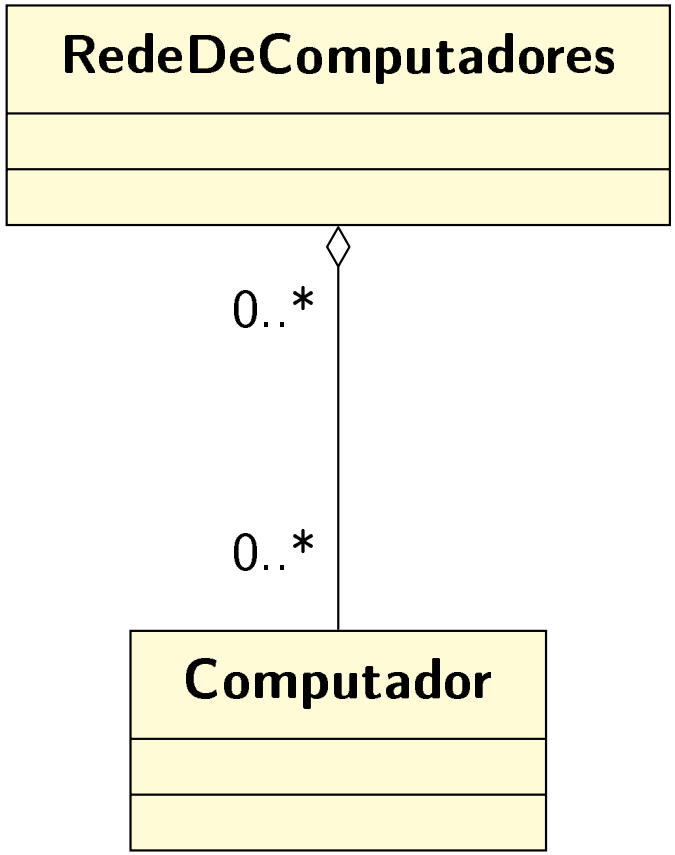
Classe parte:
É a classe cujas instâncias formam a agregação/composição

Exemplo:
Apartamento e Garagem: um apartamento pode ter garagem.

- Classe Apartamento: todo ou agregada
- Classe Garagem: parte

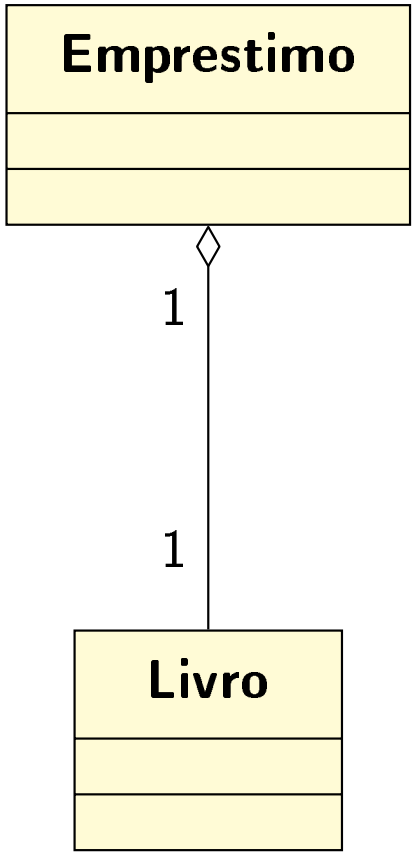

Apartamento
Garagem
Classe todo
Classe parte

Agregação: mais exemplos I



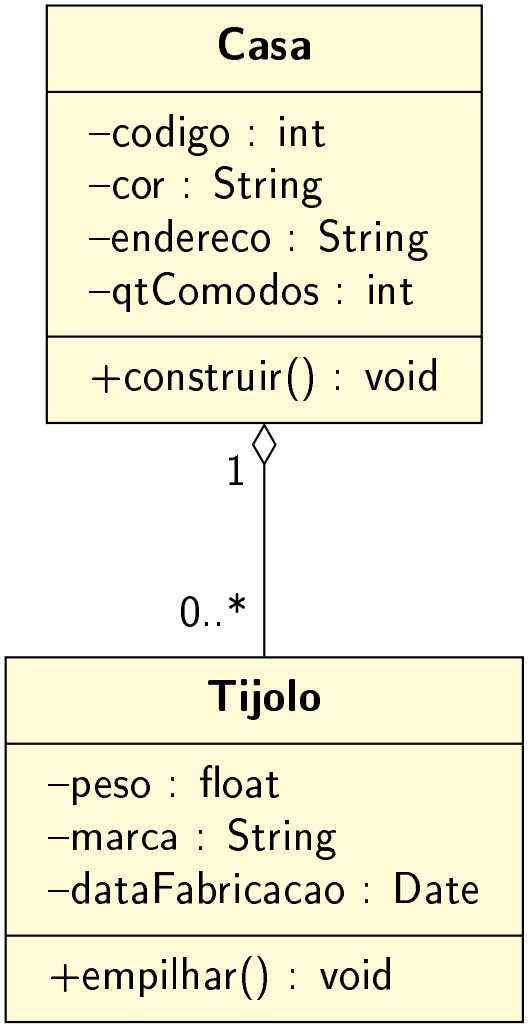
- Um computador existe independentemente de uma rede;
- Um computador pode estar ligado a mais de uma rede ao mesmo tempo.

Agregação: mais exemplos II



- Um empréstimo contém um livro, mas o livro não deixa de existir no sistema da biblioteca quando o empréstimo é concluído

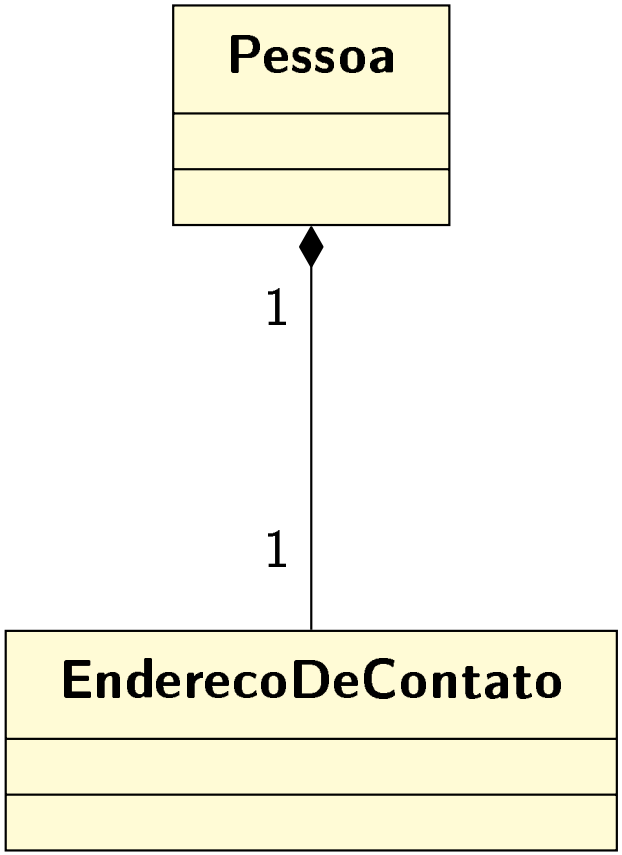
Agregação: mais exemplos III



Imagine um sistema de gerenciamento de obras e suponha as classes `Casa` e `Tijolo`:

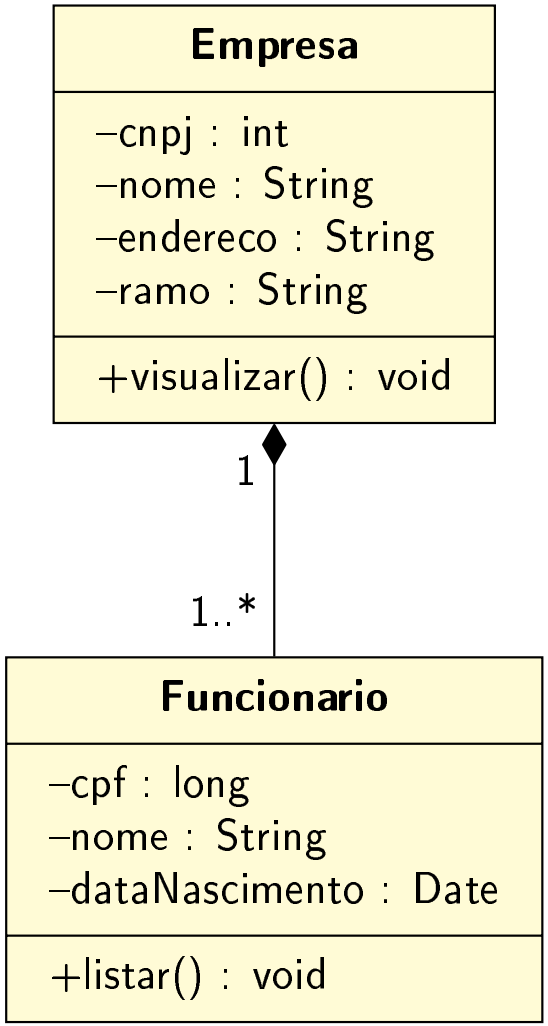
- Caso você deixe de construir uma casa, mesmo assim os tijolos poderão ser utilizados na construção de outro tipo de obra;
- Perceba que uma casa é feita de tijolos (relação **todo-parte**).

Composição: mais exemplos I



- O endereço de contato só faz sentido associado com uma pessoa;
- Se a pessoa é eliminada do sistema, não faz sentido manter o endereço de contato.

Composição: mais exemplos II



Imagine um sistema de Recursos Humanos e suponha as classes `Funcionário` e `Empresa`:

- Não faz sentido ter funcionários, se não existir uma empresa onde eles possam trabalhar;
- Se a empresa deixar de existir, automaticamente ela deixará de ter funcionários;
- Perceba que uma empresa é composta por funcionários (relação **todo-parte**).

```
class Pessoa {
    String nome;
    char sexo;
    Data dataNasc; // Data: classe
    ...
}
```

```
class Data {
    private int dia, mes, ano;
    public void alteraData(int d, int m, int a){
        dia = d;
        mes = m;
        ano = a;
    }
}
```

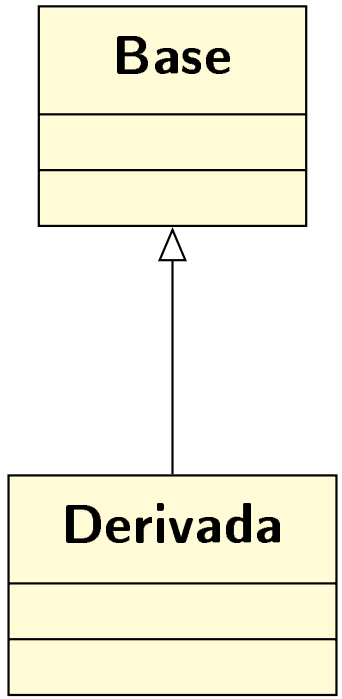
Representa relacionamentos entre classes do tipo **"é um"**.

- Também chamado de **herança**.
  Exemplo: Um cachorro é um mamífero

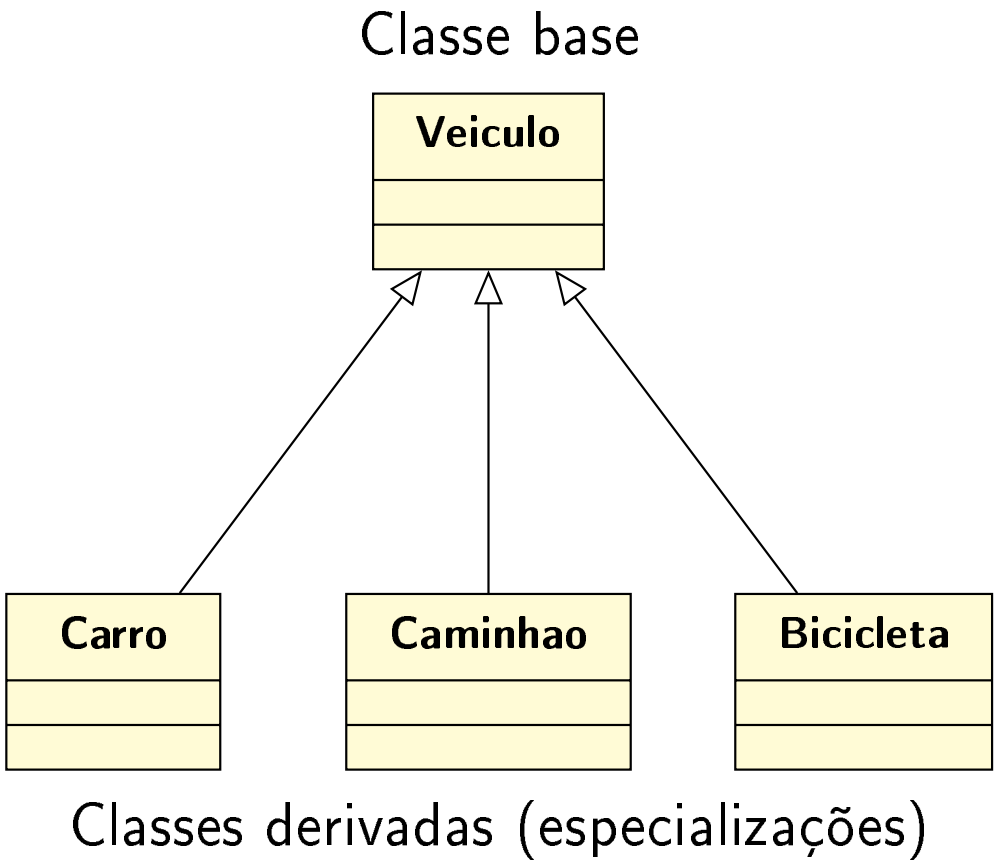
Abstração de **Generalização/Especialização**:

- A partir de duas ou mais classes, abstrai-se uma classe mais genérica;
  - Ou de uma classe geral, deriva-se outra mais específica.
- Subclasses satisfazem todas as propriedades das classes as quais as mesmas constituem especializações;
- Deve haver ao menos uma propriedade que distingue duas classes especializadas.

No diagrama de classes, a generalização é representada por uma seta do lado da classe mais geral (**classe base**).



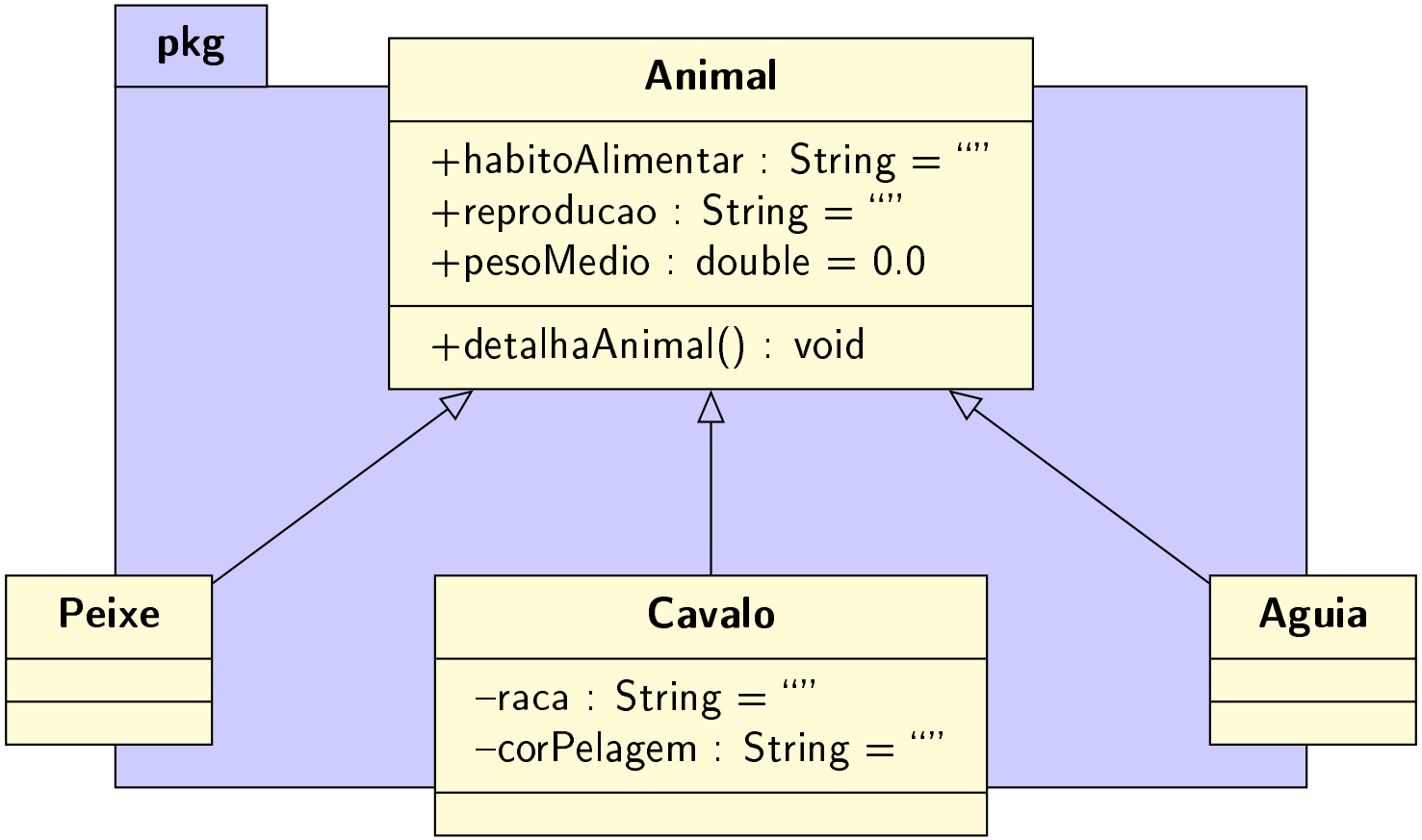
Exemplos:



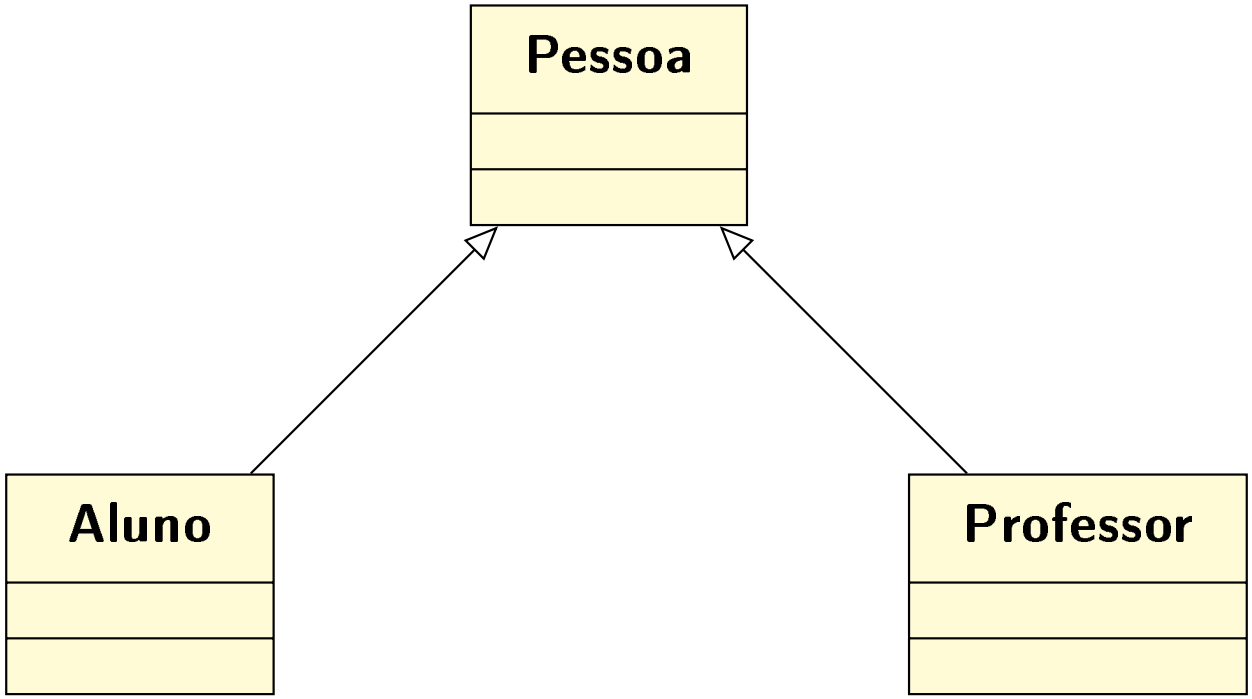
⇑
Generalização

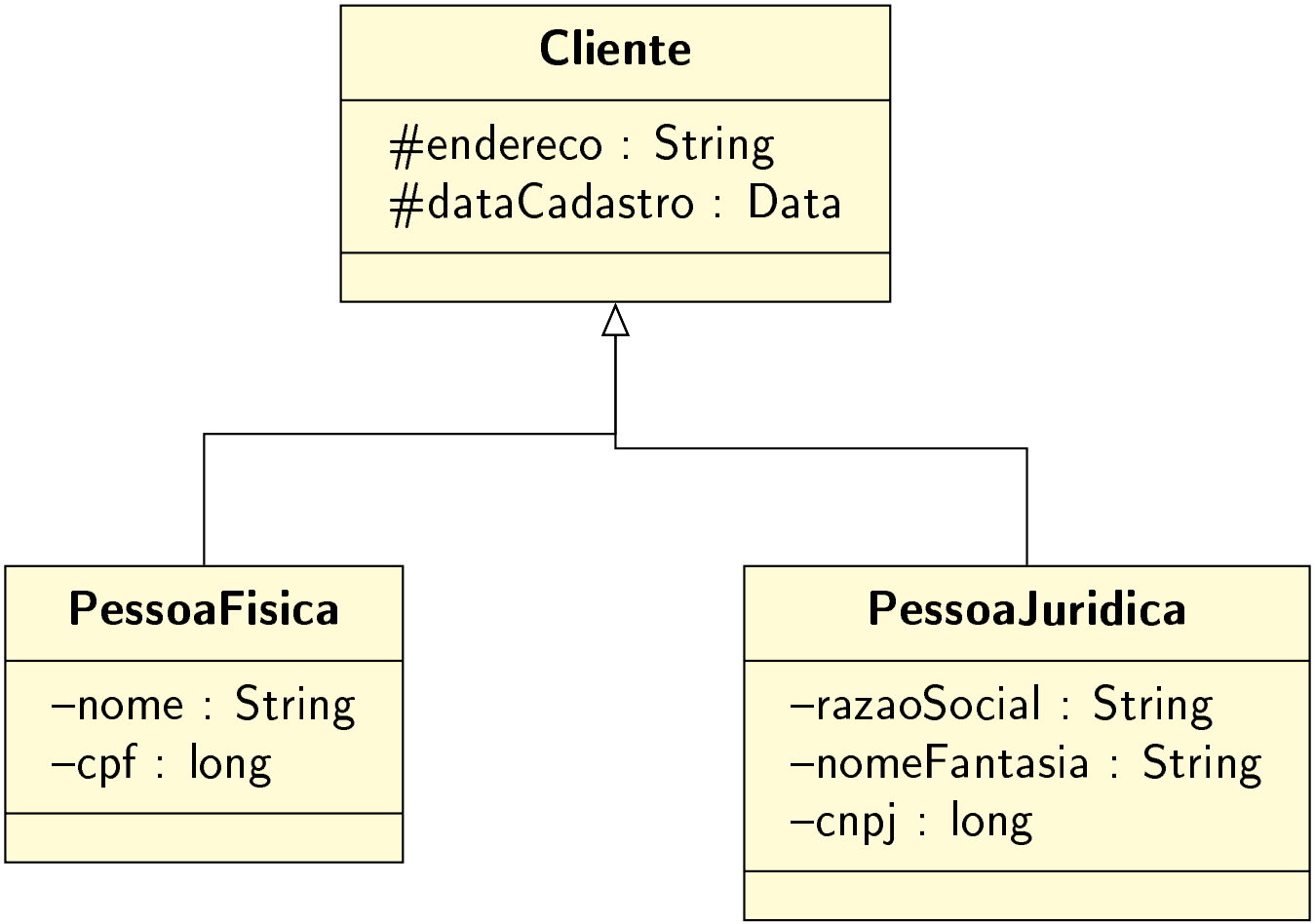
Especialização (herança)
⇓

Generalização ou herança IV

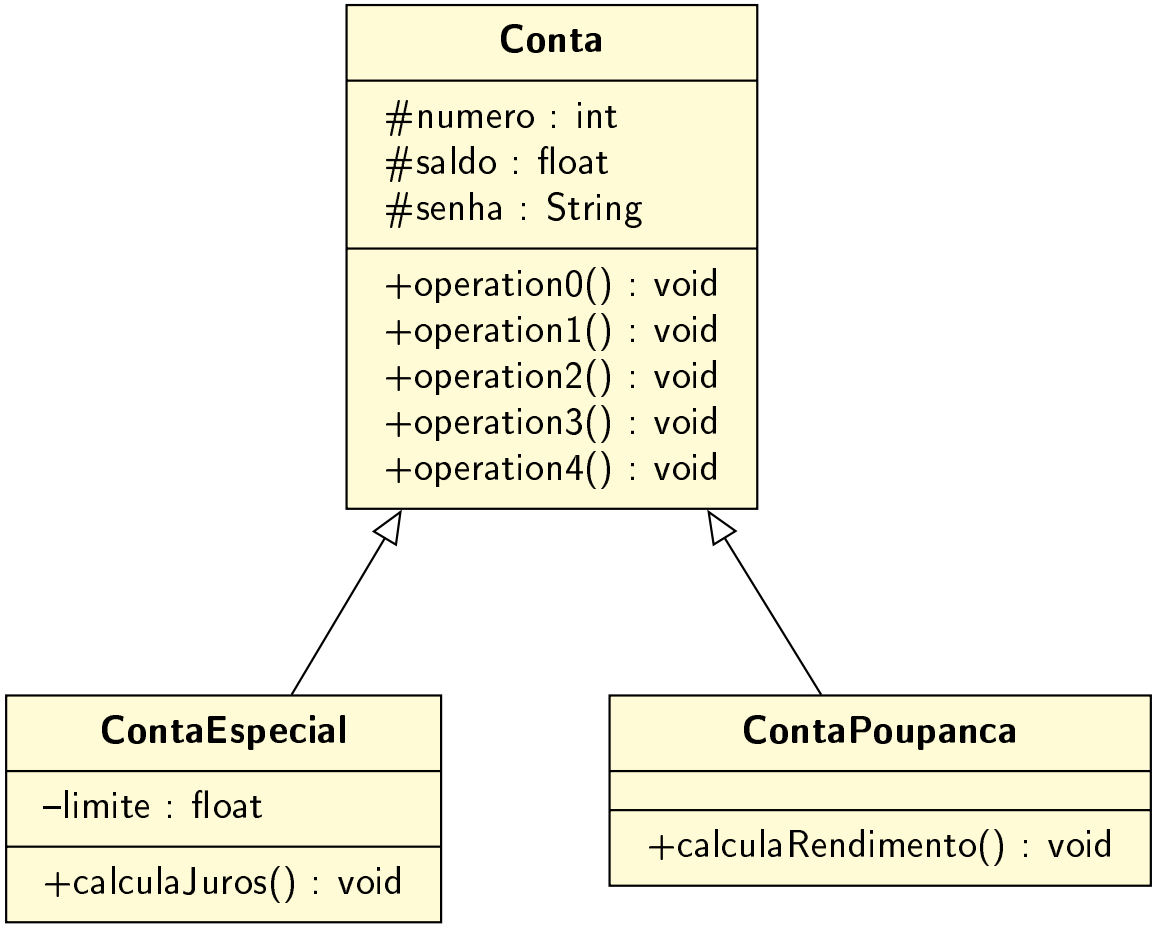


Generalização ou herança V




Cliente
#endereco : String
#dataCadastro : Data
PessoaFisica
–nome : String
–cpf : long
PessoaJuridica
–razaoSocial : String
–nomeFantasia : String
–cnpj : long

Generalização ou herança VII



- A generalização permite organizar as classes de objetos hierarquicamente;
- Ainda, consiste numa forma de **reutilização** de software:
  - Novas classes são criadas a partir de existentes, absorvendo seus atributos e comportamentos, acrescentando recursos que necessitem

Como saber qual relacionamento deve ser utilizado?

- Existem atributos ou métodos sendo aproveitados por outras classes? A subclasse "é do tipo" da superclasse?
- **Sim**: Isso é herança
- **Não**: Existe todo-parte?
  - **Sim**: A parte vive sem o todo?
    - ★ **Sim**: Isso é agregação
    - ★ **Não**: Isso é uma composição
  - **Não**: Isso é associação

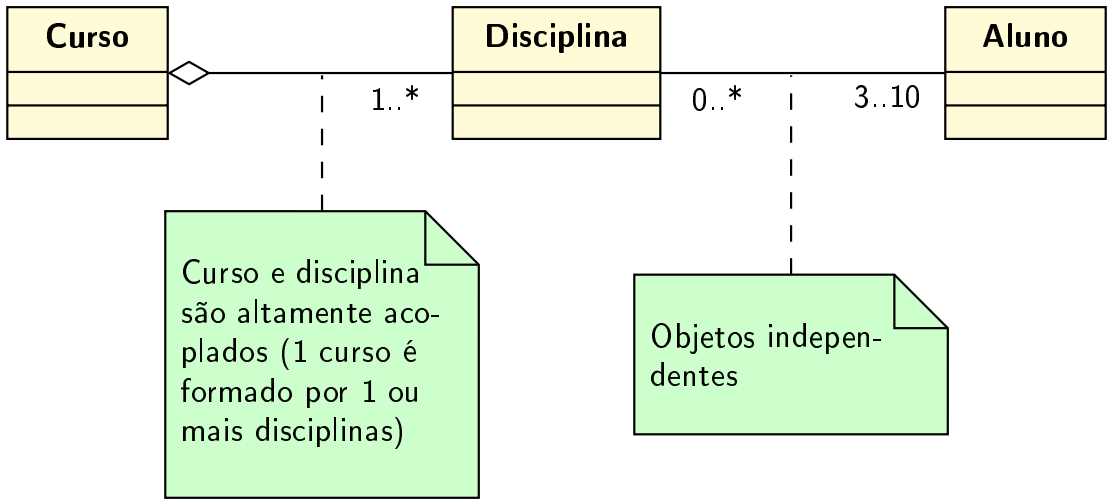
Associação vs. agregação/composição I

Se dois objetos são altamente acoplados por um relacionamento todo-parte:

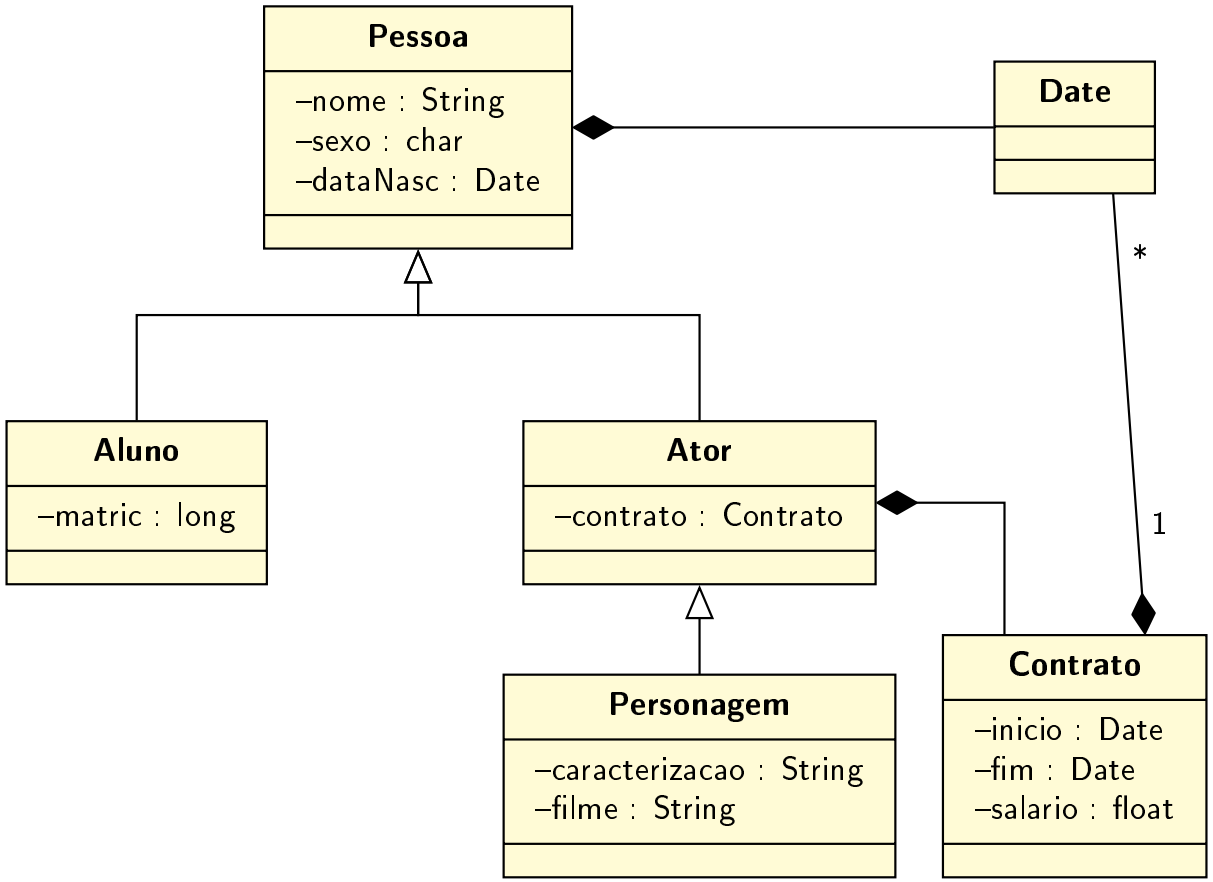
- O relacionamento é uma **agregação** ou **composição**.

Se dois objetos são usualmente considerados como independentes, mesmo eles estejam frequentemente ligados:

- O relacionamento é uma **associação**.



Exemplo:


Pessoa
–nome : String
–sexo : char
–dataNasc : Date
Date
*
1
Aluno
–matric : long
Ator
–contrato : Contrato
Personagem
–caracterizacao : String
–filme : String
Contrato
–inicio : Date
–fim : Date
–salario : float

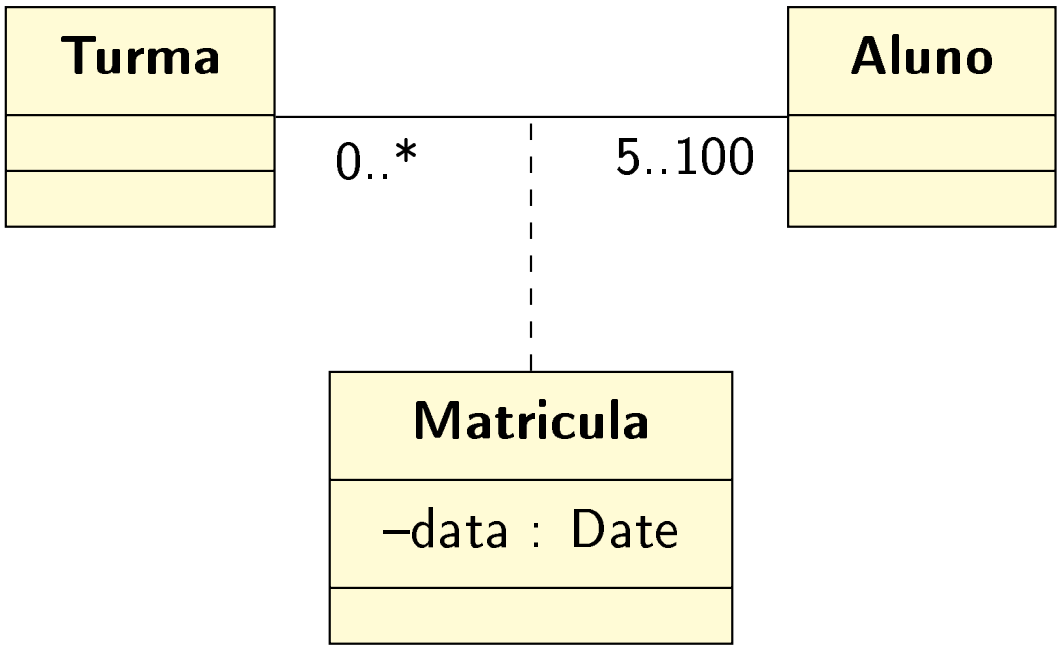
Classes de associação I

As **classes de associação** são classes que fornecem um meio para adicionar atributos e operações a associações.

- Normalmente geradas entre ocorrências que possuem multiplicidade muitos nas extremidades
- **Exemplo**, considere o relacionamento a seguir:

- Deseja-se agora acrescentar a data em que cada aluno foi adicionado à turma;

- Obviamente, esta data não é uma propriedade *nem do aluno* e *nem da turma*.
- Sendo assim, criamos uma **classe associativa**, chamada, por exemplo, de Matricula:



Exercícios I

1. Faça a modelagem em UML de um sistema bancário, relacionado à administração de contas bancárias (para cada classe defina, pelo menos, 4 atributos e 4 métodos). Em um banco há gerentes responsáveis por um grupo de clientes.
   - Um gerente poderá aceitar pedidos de abertura de conta, de empréstimo, de cartão de crédito, etc. Mas poderá decidir por oferecer os serviços, ou não.
   - Cada cliente poderá pedir serviços para um gerente: abertura de contas, empréstimo, cartão de crédito, etc. Ele também poderá ter acesso à sua conta bancária.
   - Cada conta bancária poderá oferecer serviços tais como: depositar, sacar, transferir dinheiro entre contas, pagar cartão de crédito, etc.
   - Após a modelagem, para cada classe coloque quais serviços pode solicitar das outras classes.

Exercícios II

Gerente
–nome : String –funcao : String –numeroClientes : int –cpf : long
+iniciarPedidoEmprestimo() +iniciarPedidoCartao() –liberarEmprestimo() –liberarCartao()

Cliente
–nome : String –cpf : long –salario : double –profissao : String
+atualizarSenha() +cadastrarComputador() +pedirEmprestimo() +pedirCartao()

ContaBancaria
–nomeCliente : String –tipoConta : String –validade : String –dataCriacao : String
+depositar() +sacar() +transferir() +pagarCartao()

2. Faça a modelagem em UML de um sistema de controle de cursos de informática equivalente ao módulo de matrícula de acordo com os seguintes fatos:
   - o curso pode ter mais de uma turma, no entanto, uma turma se relaciona exclusivamente com um único curso.
   - uma turma pode ter diversos alunos matriculados, no entanto uma matrícula refere-se exclusivamente a uma determinada turma. Cada turma tem um número mínimo de matriculas para iniciar o curso.
   - um aluno pode realizar muitas matrículas, mas cada matrícula refere-se exclusivamente a uma turma específica e a um único aluno.

Exercícios IV

3. Faça a modelagem em UML de um sistema de reserva para uma empresa aérea (para cada classe defina, pelo menos, 4 atributos e 4 métodos).
   - Cada voo deverá estar cadastrado no sistema, pois as reservas serão relacionadas a eles. Cada voo pode informar o número de assentos livres, sua tripulação, reservar acento, etc
   - Operadores são funcionários da empresa responsáveis pela operacionalização das reservas. Os operadores fazem as reservas, as cancelam, informam sobre possíveis atrasos, etc
   - Os clientes podem pedir reservas nos voos, podem cancelar reservas, podem pagá-las de forma adiantada, etc

   Após a modelagem, para cada classe coloque quais serviços pode solicitar das outras classes.

Referências

1. Addison-BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML, Guia do Usuário*. Rio de Janeiro: Campus, 2000.
2. FOWLER, M. *UML Essencial*, 2a Edição. Porto Alegre: Bookman, 2000.
3. LARMAN, C. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos*. Porto Alegre: Bookman, 2001.

Os slides dessa apresentação foram cedidos por:

- Graça Marietto e Francisco Zampirolli, UFABC
- Profa Katti Faceli, UFSCar/Sorocaba
- Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem: Renato Pimentel, FACOM/UFU

No mundo real, através da Genética, é possível herdarmos certas características de nossos pais.
De forma geral, herdamos atributos e comportamentos de nossos pais.

Da mesma forma, em OO nossas classes também podem herdar características (atributos e comportamentos) de uma classe já existente. Chamamos este processo de **herança**.

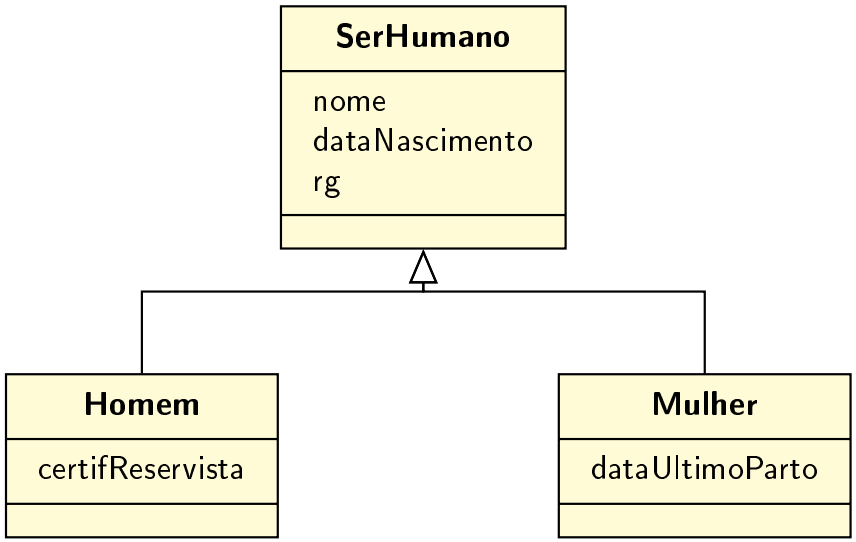
Herança permite a criação de classes com base em uma classe já existente

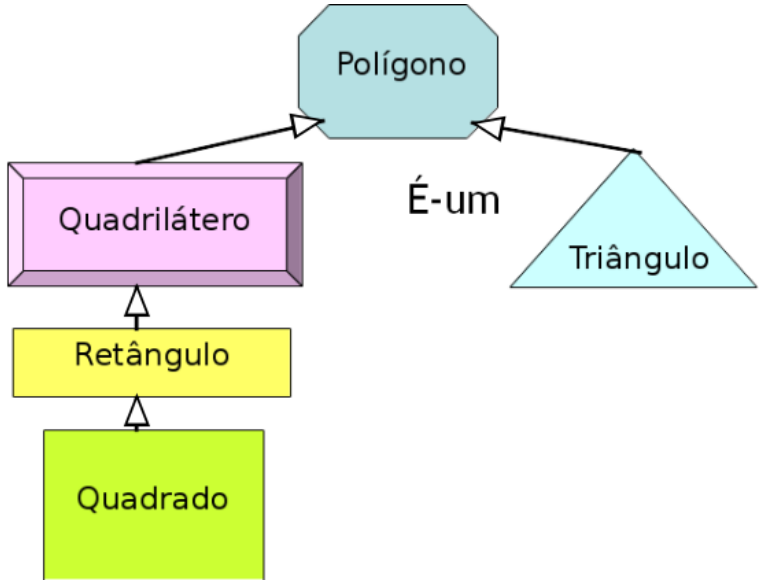
Objetivo: proporcionar o reuso de software

Herança é a capacidade de reusar código pela especialização de soluções genéricas já existentes

- A ideia na herança é *ampliar* a funcionalidade de uma classe

Todo objeto da subclasse também é um objeto da superclasse, mas **não** vice-versa.

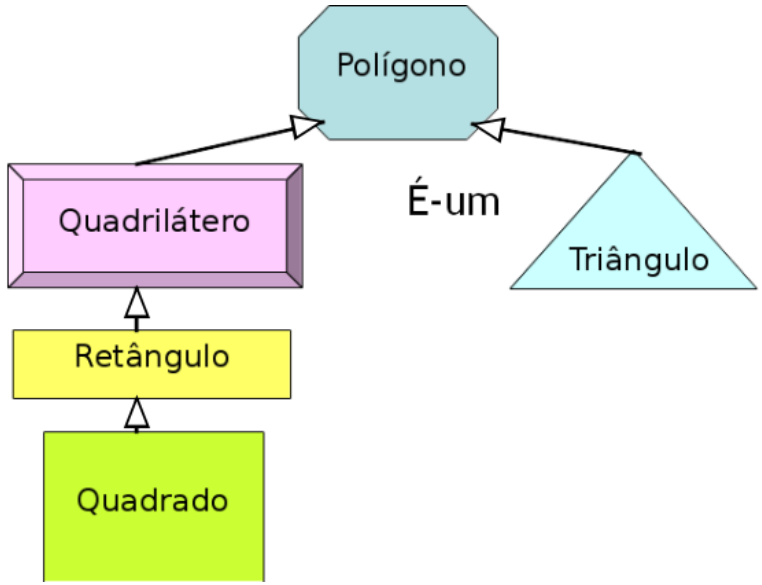




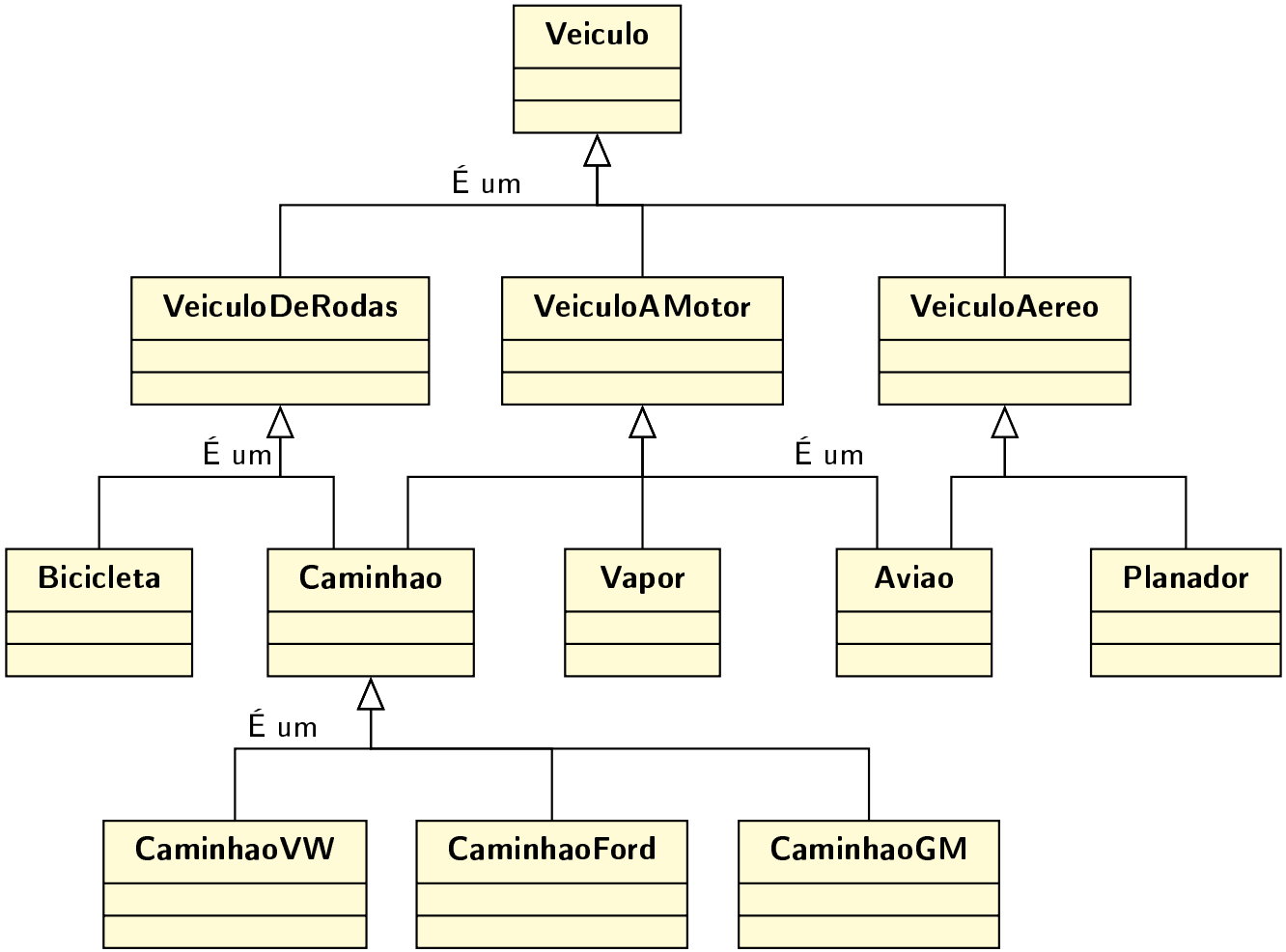
A representação gráfica do conceito de herança, na linguagem UML (*Unified Modeling Language*), é definida por retas com setas apontando para a *classe-mãe*.

Herança representa um relacionamento de generalização entre classes:

- **É um**;
- **É um tipo de**.

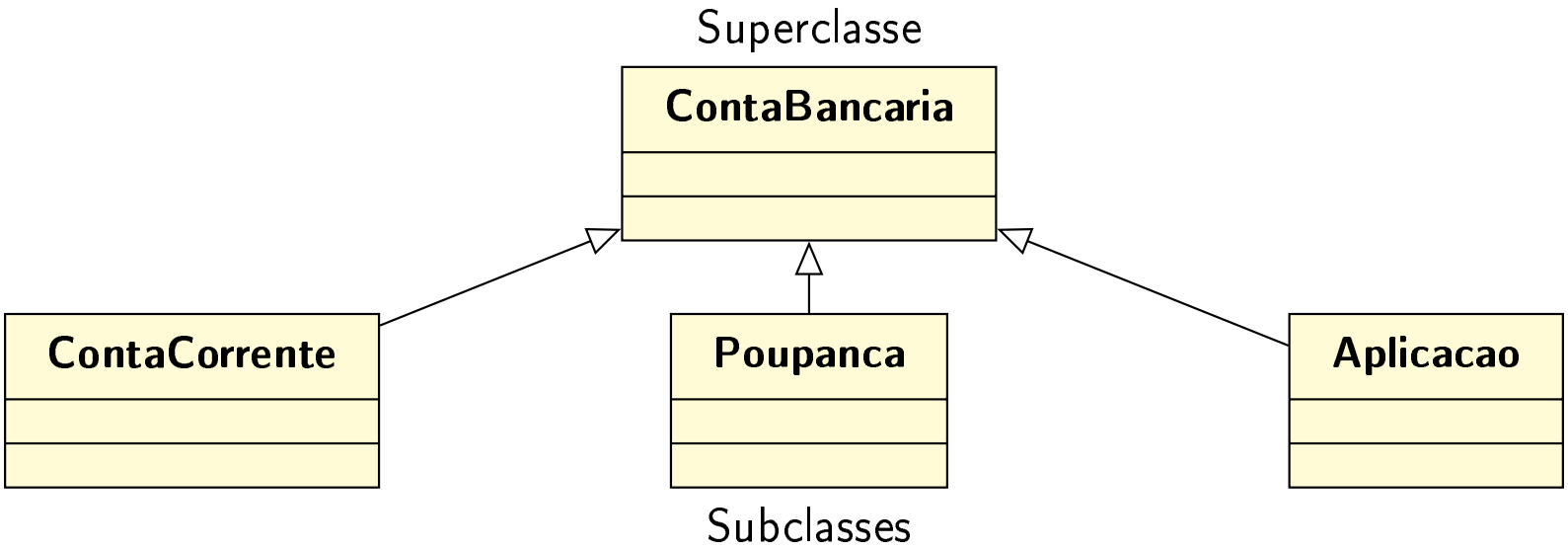


Herança V


Veiculo
É um
VeiculoDeRodas
VeiculoAMotor
VeiculoAereo
É um
É um
Bicicleta
Caminhao
Vapor
Aviao
Planador
É um
CaminhaoVW
CaminhaoFord
CaminhaoGM

Terminologia:

- **Classe mãe**, **superclasse**, **classe base**: A classe mais geral, a partir da qual outras classes herdam características (atributos e métodos);
- **Classe filha**, **subclasse**, **classe derivada**: A classe mais especializada, que herda características de uma classe mãe.

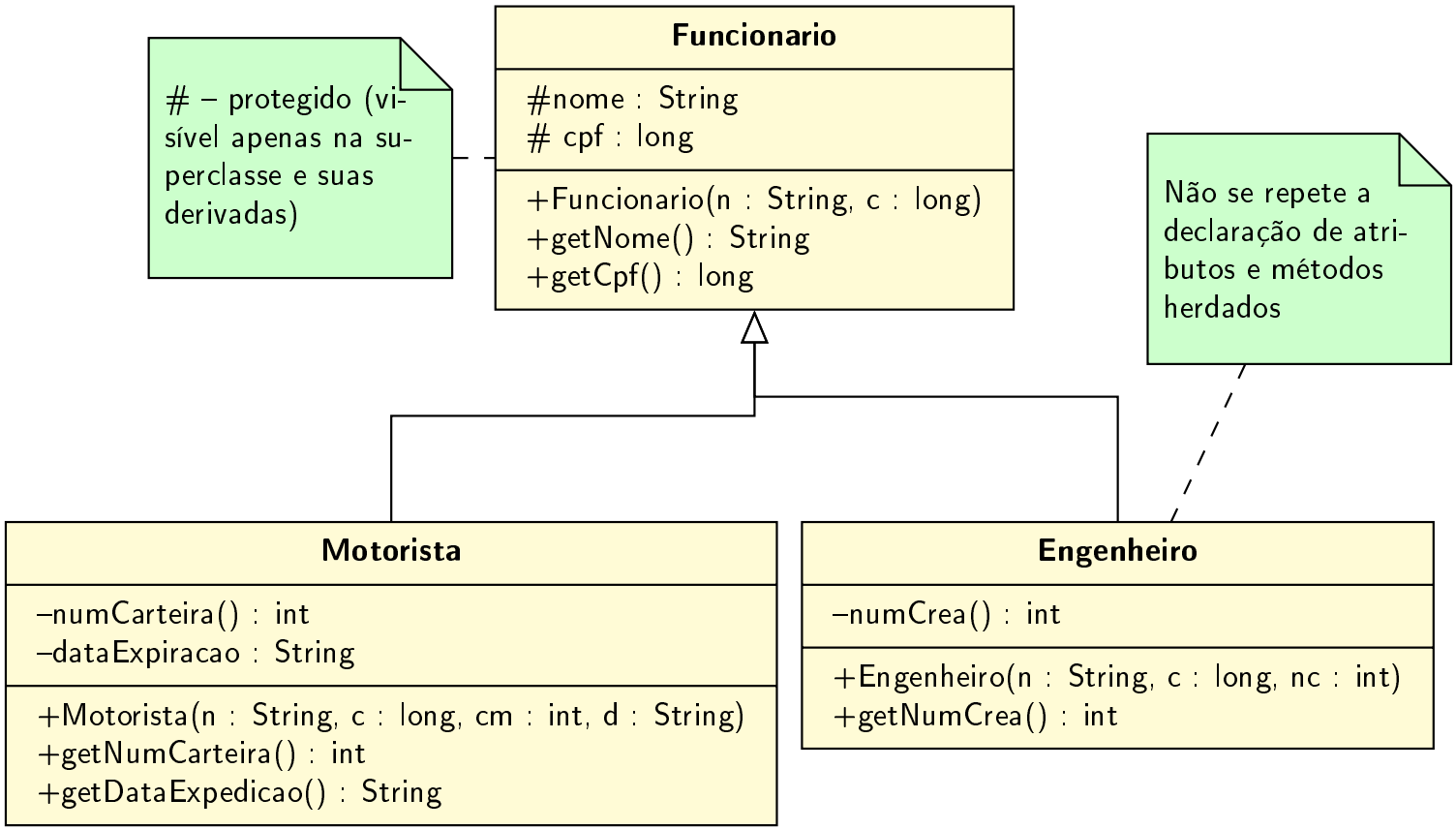

Superclasse
ContaBancaria
ContaCorrente
Poupanca
Aplicacao
Subclasses

Uma subclasse **herda** atributos e métodos de sua superclasse, podendo possuir, no entanto, membros que lhe são próprios.

Acerca dos membros herdados pela subclasse:

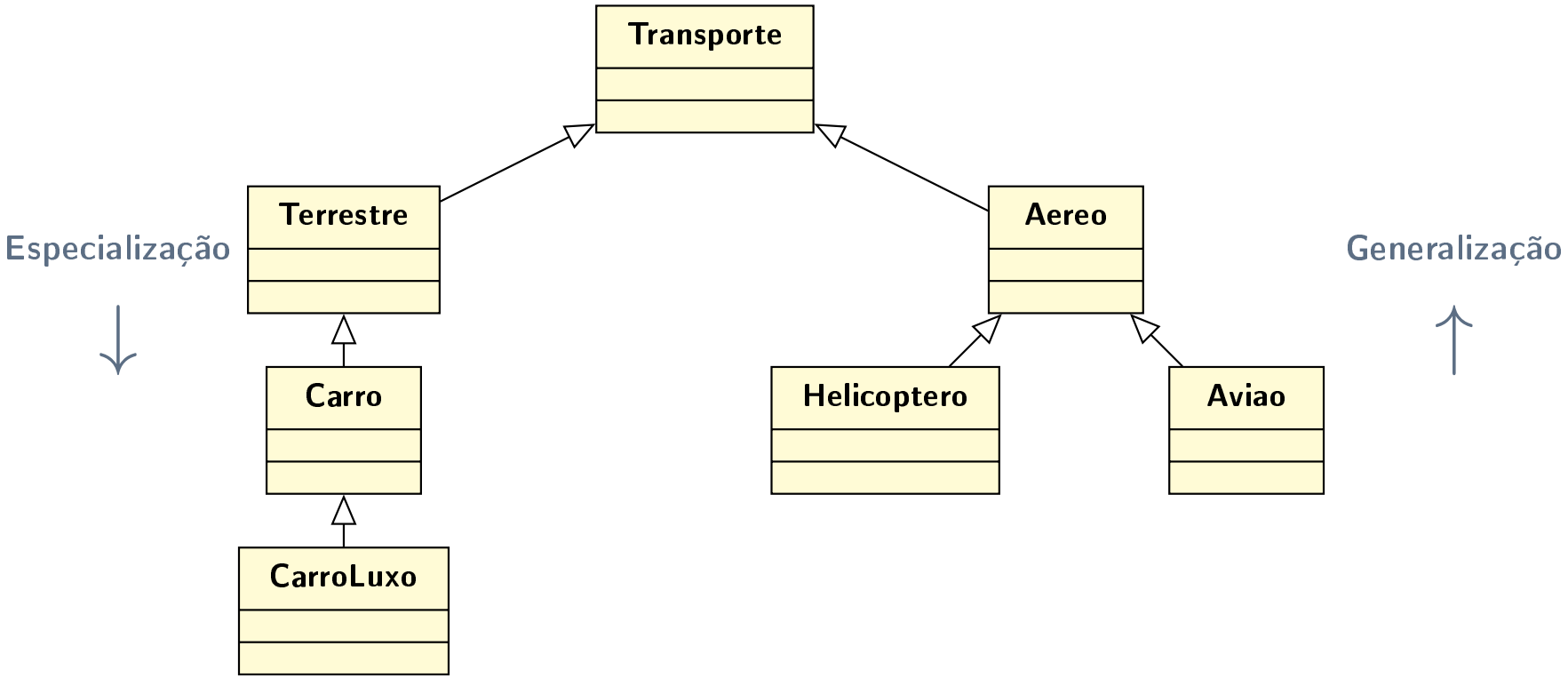
- Tratados de forma semelhante a qualquer outro membro da subclasse;
- Nem todos os membros da superclasse são acessíveis pela subclasse (encapsulamento) – se membro da superclasse é encapsulado como `private`, subclasse não o acessa;
- Na superclasse, tornam-se seus membros acessíveis *apenas* às subclasses usando o modificador de acesso `protected` (diagramas de classe UML: #) do Java.

Exemplo:

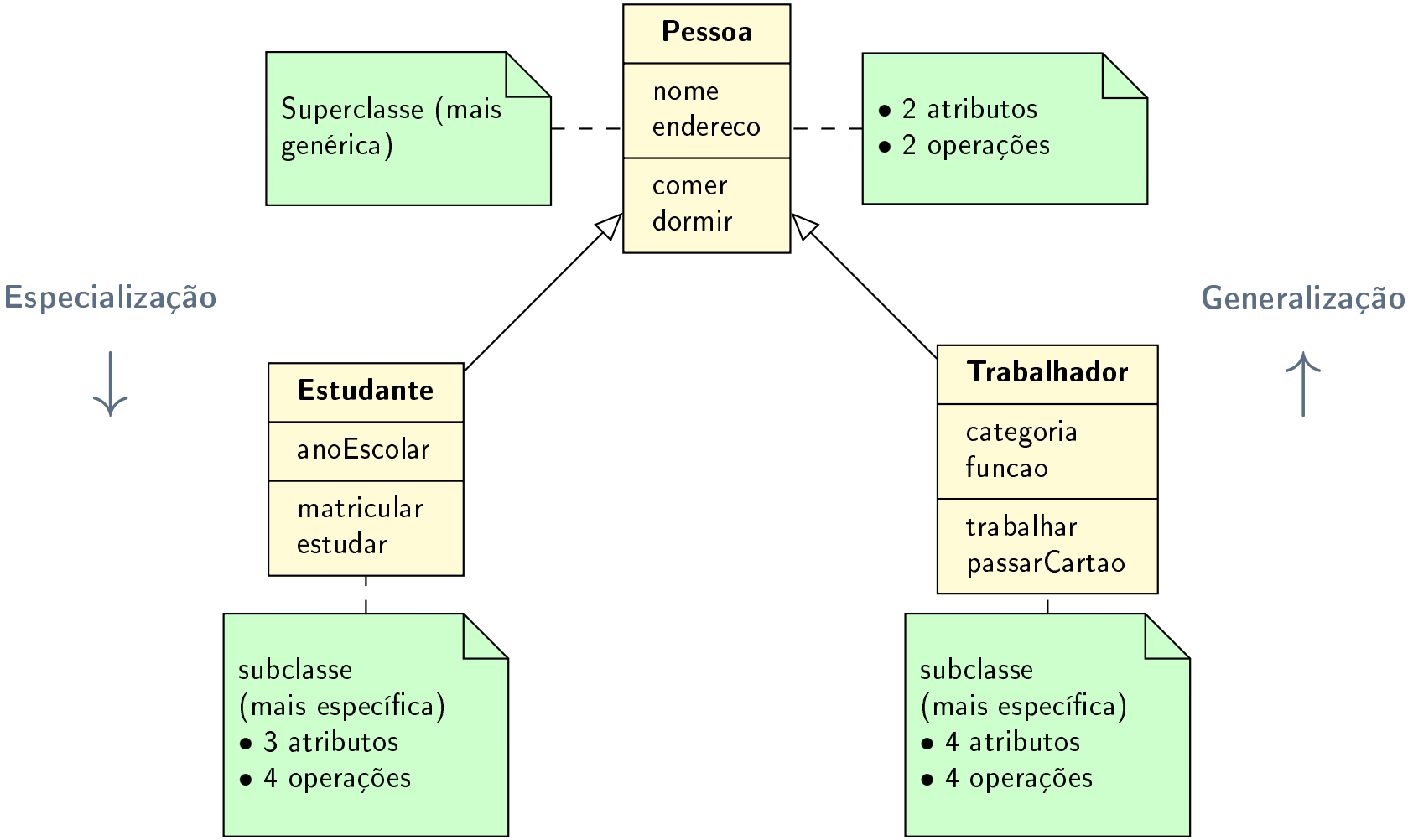

– protegido (visível apenas na superclasse e suas derivadas)
Funcionario
#nome : String
cpf : long
+Funcionario(n : String, c : long)
+getNome() : String
+getCpf() : long
Não se repete a declaração de atributos e métodos herdados
Motorista
–numCarteira() : int
–dataExpiracao : String
+Motorista(n : String, c : long, cm : int, d : String)
+getNumCarteira() : int
+getDataExpedicao() : String
Engenheiro
–numCrea() : int
+Engenheiro(n : String, c : long, nc : int)
+getNumCrea() : int

Especialização vs. generalização I

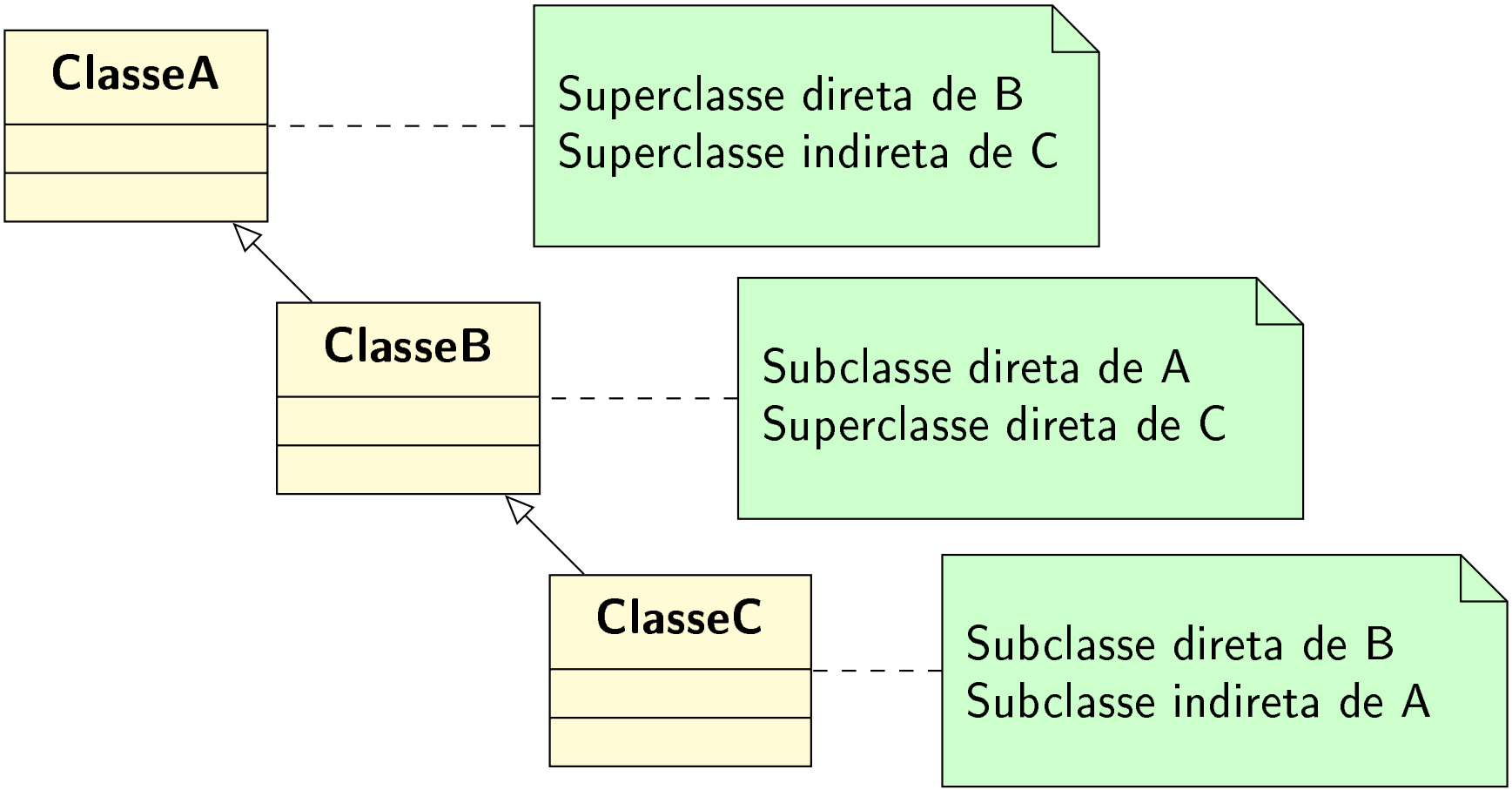
Formas diferentes de se pensar a hierarquia de classes, e também de se modelar sistemas em POO.



Especialização vs. generalização II



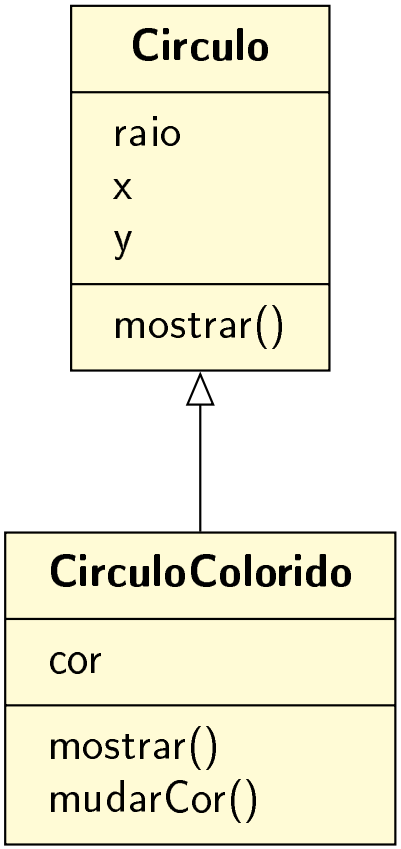
O processo de herança pode ser repetido *em cascata*, criando **várias gerações** de classes.



Uma subclasse pode diferenciar-se de sua classe mãe:

- Com *operações adicionais*, diferentes das herdadas;
- *Acrescentando* atributos adicionais;
- *Sobrepondo* comportamentos existentes, oferecidos pela superclasse, contudo inapropriados para a nova classe (**overriding**)

Exemplo:



- A classe `CirculoColorido` herda da classe `Circulo` os atributos `raio`, `x` e `y`;
- Entretanto, define um novo atributo **cor**, **redefine** o método `mostrar()`, e implementa o método `mudarCor()`.

Sobrescrita de métodos (*overriding*) I

A *redefinição* de um método herdado pela subclasse é feita ao definirmos na mesma um método *com mesmo nome, tipo de retorno e número de argumentos* (procedimento denominado **sobrescrita** ou **overriding**).

- **Ideia:** reimplementar o método definido previamente na superclasse de forma diferente, mais apropriada para a subclasse em questão.
- Ou seja, na classe derivada pode(m) haver método(s) com o mesmo nome de métodos da classe mãe, mas, por exemplo, com *funcionalidades diferentes*.

Sobrescrita e sobrecarga

Se o nome do método na subclasse for o mesmo de outro na classe mãe, mas os parâmetros forem diferentes, então ocorrerá uma **sobrecarga**, e não uma sobrescrita.

Outra observação importante:

- O modificador de acesso do método da subclasse pode **relaxar o acesso**, mas não o contrário
  - Ex.: um método `protected` na superclasse pode se tornar `public` na subclasse, mas não `private`.

Por fim, uma subclasse não pode sobrescrever um *método de classe* (isto é, `static`) da superclasse.

- **Reutilização** do código;
- Modificação de uma classe sem mudanças na classe original;
- É possível modificar uma classe para criar uma nova, de comportamento apenas ligeiramente diferente;
- Pode-se ter diversos objetos que executam ações diferentes, mesmo possuindo a **mesma origem**.

Um exemplo prático I

Os professores de uma universidade dividem-se em 2 categorias:

- Professores em dedicação exclusiva (DE)
- Professores horistas

Considerações:

- Professores DE possuem um salário fixo para 40 horas de atividade semanais;
- Professores horistas recebem um valor por hora;
- O cadastro de professores desta universidade armazena o nome, idade, matrícula e informação de salário.

Modelagem:

ProfDE
–nome : String –matricula : String –cargaHoraria : int –salario : float
+ProfDE(n : String, m : String, i : int, s : float) +getNome() : String +getMatricula() : String +getCargaHoraria() : int +getSalario() : float

ProfHorista
–nome : String –matricula : String –cargaHoraria : int –salarioHora : float
+ProfHorista(n : String, m : String, t : int, s : float) +getNome() : String +getMatricula() : String +getCargaHoraria() : int +getSalario() : float

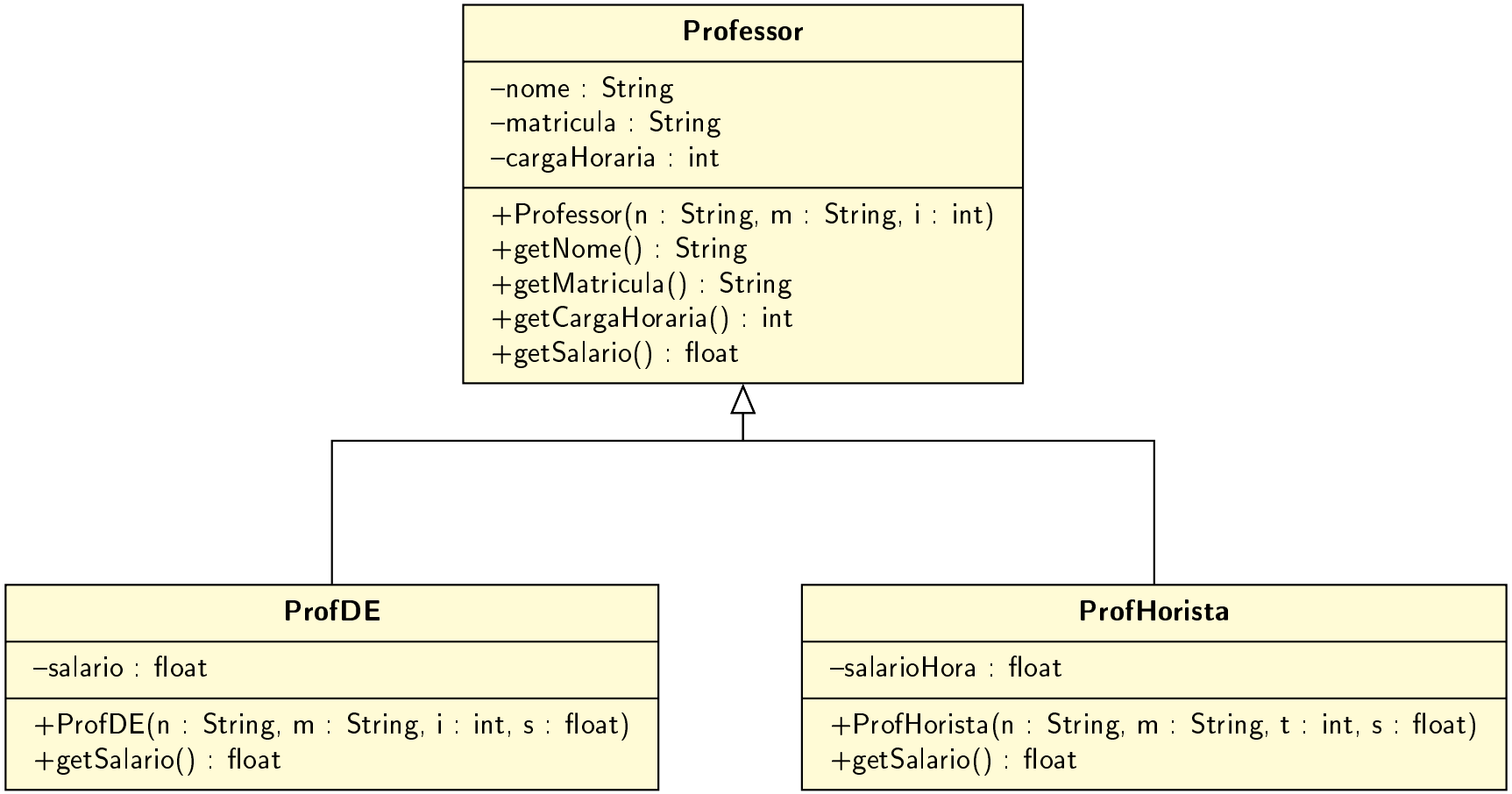
Análise:

- As classes têm alguns atributos e métodos iguais;
- O que acontece se precisarmos alterar a representação de algum atributo, como por exemplo, o número de matrícula para inteiros ao invés de uma `String`?
  - Será necessário alterar os construtores e os métodos getMatricula() nas duas classes, o que é ruim para a programação: **código redundante**.
- Como resolver? **Herança**.

Sendo assim:

- Cria-se uma classe `Professor`, que contém os *membros* – atributos e métodos – comuns aos dois tipos de professor;
- A partir dela, cria-se duas novas classes, que representarão os professores horistas e DE;
- Para isso, essas classes deverão "herdar" os atributos e métodos declarados pela classe "pai", `Professor`.

Um exemplo prático V



A palavra this é usada para referenciar membros do objeto em questão.

- É obrigatória quando há ambiguidades entre variáveis locais e de instância (atributos).

super, por sua vez, se refere à superclasse.

```
class Numero {
    public int x = 10;
}
```

```
class OutroNumero extends
      Numero {
    public int x = 20;
    public int total() {
        return this.x +
            super.x;
    }
}
```

`super()` e `this()` – *note os parênteses* – são usados **somente nos construtores**.

- `this()`: para chamar outro construtor, na mesma classe.

Exemplo:

```
public class Livro {
    private String titulo;
    public Livro() {
        this.titulo = "Sem titulo";
    }
    public Livro(String titulo) {
        this.titulo = titulo;
    }
}
```

Reimplementando:

```
1 public class Livro {
2     private String titulo;
3     public Livro() {
4         this("Sem titulo");
5     }
6     public Livro(String titulo) {
7         this.titulo = titulo;
8     }
9 }
```

- `super()`: para chamar construtor da classe base a partir de um construtor da classe derivada.

Exemplo:

```
1 class Ave extends Animal {
2     private int altura;
3     Ave() {
4         super();
5         altura = 0.0; // ou this.altura = 0.0;
6     }
7 }
```

Algumas regras para uso de `super()`:

- Construtores da classe mãe são chamados pela palavra reservada `super`, seguida pelos argumentos a serem passados para o construtor entre parênteses.
  - Se não houver argumentos, basta usar `super()`.
- Construtores de superclasses somente poder ser invocados de dentro de construtores de subclasses, e **na primeira linha de código** dos mesmos.
- Se não houver, no construtor da subclasse, uma chamada explicita ao construtor da superclasse, o construtor sem argumento é chamado por padrão
  - Se não houver construtor sem parâmetros na superclasse, o compilador apontará erro.

Vimos anteriormente um exemplo de uso de `super` para acessar um atributo da superclasse.
A palavra reservada `super` também pode ser usada para acessar métodos da superclasse. Algumas considerações:

- Outros métodos podem ser chamados pela palavra-chave `super` seguida de um ponto, do nome do método e argumento(s), se existente(s), entre parênteses:
  `super.nomeDoMetodo( [argumento(s)] );`
- Se a implementação do método for a mesma para a super e a subclasse – ou seja, não for uma sobrescrita) – então instâncias da subclasse podem chamar diretamente o método como se fosse de si próprias.

Faça os diagramas de classes a partir das descrições:

1. Defina a classe `Produto`.
   - Os atributos de um produto são: código, descrição e quantidade, com a visibilidade protegida;
   - O construtor deve receber todos atributos por parâmetro;
   - A classe deve oferecer rotinas tipo acessoras (*getters*) para todos os campos;
   - Deve oferecer uma rotina onde se informa certa quantidade a ser retirada do estoque e outra onde se informa uma certa quantidade a ser acrescida ao estoque;
   - A rotina onde se informa uma quantidade a ser retirada do estoque deve retornar a quantidade que efetivamente foi retirada (para os casos em que havia menos produtos do que o solicitado).

2. Defina a classe `ProdutoPerecivel`.
   - Esta deve ser **derivada** de Produto;
   - Possui um *atributo extra* que guarda a data de validade do produto;
   - As rotinas através das quais se informa as quantidades a serem retiradas ou acrescidas do estoque devem ser alteradas:
     - ★ A rotina de retirada deve receber também por parâmetro a data do dia corrente;
     - ★ Se os produtos já estiverem armazenados há mais de 2 meses a rotina deve zerar o estoque e devolver 0, pois produtos vencidos são descartados;
     - ★ A rotina de acréscimo no estoque só deve acrescentar os novos produtos caso o estoque esteja zerado, de maneira a evitar misturar produtos com prazos de validade diferenciados.

3. Defina a classe `ProdutoPerEsp`.
   - Esta é derivada de `ProdutoPerecivel`;
   - Oferece uma rotina de impressão de dados capaz de imprimir uma nota de controle onde consta o código, a descrição, a quantidade em estoque e a data de validade do produto.
4. Defina a classe `ProdutoComPreco`.
   - Esta é derivada de `Produto`;
   - Deve possuir campos para armazenar o preço unitário do produto;
   - A classe deve oferecer rotinas para permitir obter e alterar o preço unitário (sempre positivo).

5. Defina a classe `Estoque`.
   - Esta mantém uma lista com os produtos em estoque (do tipo `ProdutoComPreco`);
   - A classe deve ter métodos para cadastrar e consultar produtos, inseri-los e retirá-los do estoque, bem como para informar o custo total do estoque armazenado.

Referências

1. Addison-BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML, Guia do Usuário*. Rio de Janeiro: Campus, 2000.
2. FOWLER, M. *UML Essencial*, 2a Edição. Porto Alegre: Bookman, 2000.
3. LARMAN, C. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos*. Porto Alegre: Bookman, 2001.

Os slides dessa apresentação foram cedidos por:

- Graça Marietto e Francisco Zampirolli, UFABC
- Profa Katti Faceli, UFSCar/Sorocaba
- Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU

Termo originário do grego *poly* (muitas) + *morpho* (formas).

O **polimorfismo** em POO é a habilidade de objetos de uma ou mais classes em responder a uma mesma mensagem de *formas diferentes*.

Polimorfismo II

- Métodos com mesmo nome, mas implementados de maneira diferente.
- Permite obter códigos genéricos:
  - Processar diversos tipos de dados;
  - Processar os dados de formas distintas;
  - Podem fazer um mesmo objeto ter comportamentos diferentes para uma mesma ação/solicitação.

O polimorfismo pode ocorrer de duas maneiras:

- **Sobrecarga** (*Overloading*)
- **Sobreposição** ou **sobreescrita** (*Overriding*)

Alguns autores não classificam a sobrecarga como um tipo de polimorfismo.

Sobrecarga I

Permite que um método seja definido com diferentes *assinaturas* e diferentes implementações.

- **Assinatura**: relacionada ao *número* e *tipo* dos parâmetros.

Resolvido pelo compilador em tempo de compilação:

- A assinatura diferente permite ao compilador dizer qual dos *sinônimos* será utilizado.

Exemplo: quando definimos diferentes construtores em uma classe, cada um recebendo parâmetros diferentes.

Atenção

Mudar o nome dos parâmetros não é uma sobrecarga, o compilador diferencia o tipo, e não o nome dos parâmetros.

Exemplo: métodos
`f(int a, int b)` e
`f(int c, int d)`
numa mesma classe resultam em *erro de redeclaração*.

Como dito, as assinaturas devem ser diferentes. O que é a assinatura?
A **assinatura** de um método é composta pelo **nome do método** e pelos **tipos dos seus argumentos**, *independente dos nomes dos argumentos e do valor de retorno da função*.
Ex.: 2 assinaturas iguais:

```
float soma(float a, float b);
void soma(float op1, float op2);
```

Ex.: 2 assinaturas diferentes:

```
float soma(float a, float b);
double soma(double a, double b);
```

Sobrecarga IV

É implementada, normalmente, para métodos que devem executar operações semelhantes, usando uma lógica de programação diferente para diferentes tipos de dados.

```
1  public class Funcoes{
2      public static int quadrado( int x ) {
3          return x * x;
4      }
5
6      public static double quadrado( double y ) {
7          return y * y;
8      }
9  }
10 // ...
11 System.out.println("2 ao quadrado: " + Funcoes.quadrado(2));
12 System.out.println("PI ao quadrado: " + Funcoes.quadrado(Math.PI));
```

Em muitos casos é necessário criar métodos que precisam de mais ou menos parâmetros, ou até precisem de parâmetros de tipos diferentes.

```
1 Fracao f1 = new Fracao();  // exemplo início curso: num=0,den=1
2 Fracao f2 = new Fracao(f1); // copy constructor: copia conteúdo de
      f1
3 Fracao f3 = new Fracao(5);  // num=5, den=1 (inteiro)
4 Fracao f4 = new Fracao(5,2);
5
6 // Exercício: como seria a implementação dos construtores das
      linhas 2 e 3 dentro da classe Fracao?
```

Conceito já visto em **herança**:

- Permite a redefinição do funcionamento de um método herdado de uma *classe base*.
- A *classe derivada* tem uma função **com a mesma assinatura** da classe base, mas **funcionamento diferente**;
- O método na classe derivada **sobrepõe** a função na classe base.

Polimorfismo estático × Polimorfismo dinâmico

Sobrescrita: **polimorfismo dinâmico** – envolve 2 classes (classe derivada herda e redefine método da classe base); **Polimorfismo estático** – métodos com mesmo nome e assinaturas diferentes na mesma classe (sobrecarga).

Indicações para uso da sobrescrita:

- A implementação do método na classe base não é adequada na classe derivada;
- A classe base não oferece implementação para o método, somente a declaração;
- A classe derivada pretende estender as funcionalidades da classe base.

Exemplo: considere as seguintes classes:

```
public class Figura {
    ...
    public void desenha(Graphics g) {
        ...
    }
}
```

```
public class Retangulo extends Figura {
...
    public void desenha(Graphics g) {
        g.drawRect(x, y, lado, lado2);
    }
}
```

```
public class Circulo extends Figura {
    ...
    public void desenha(Graphics g) {
        g.drawCircle(x, y, raio);
    }
}
```

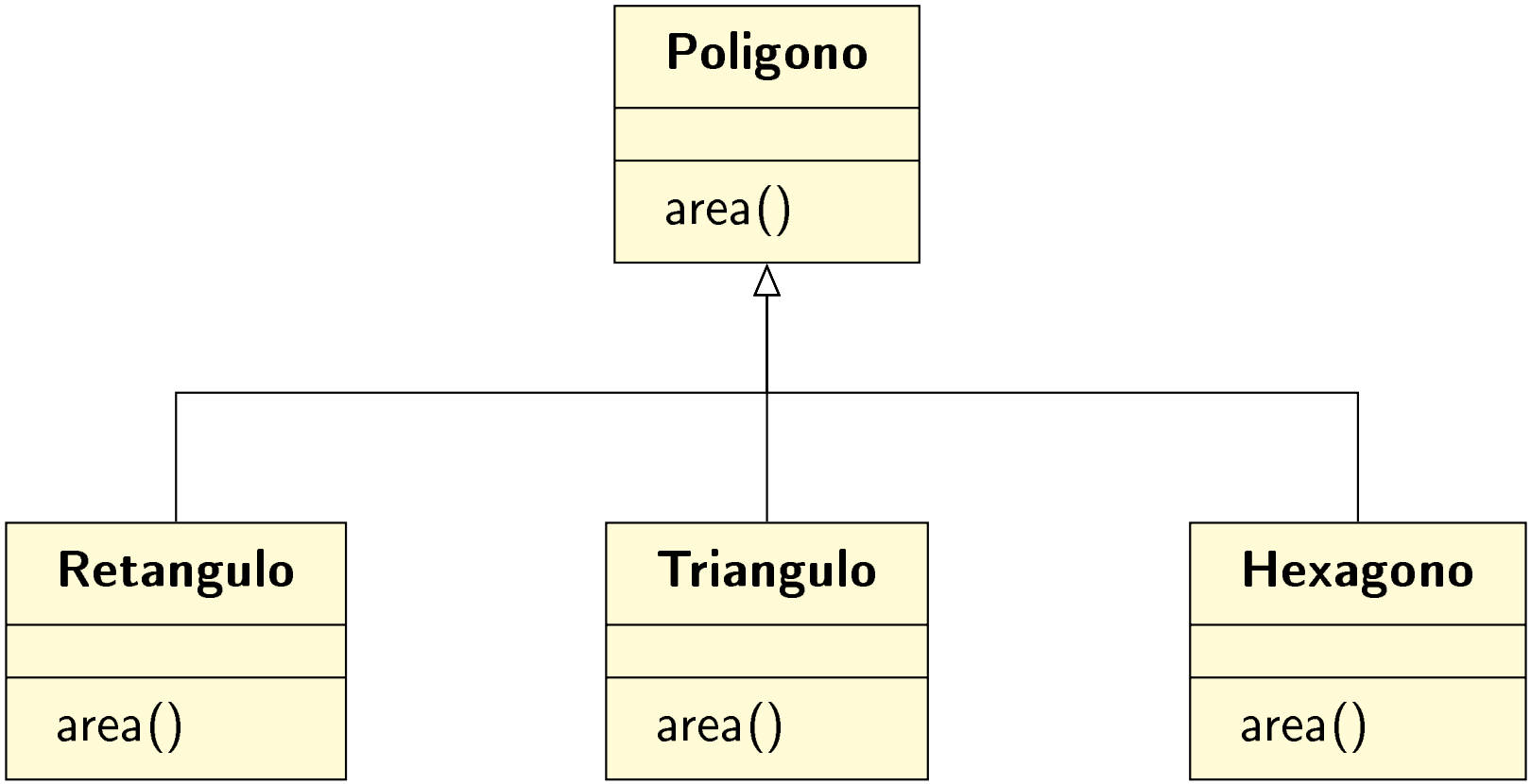
Sobrescrita VI

Na classe principal:

```
...
for (int i = 0; i < desenhos.size(); ++i) {
    Figura x = desenhos[i];
    x.desenha(g);
}
...
```

Na ocasião desta chamada, será decidido *automaticamente* qual implementação será invocada, dependendo do objeto: esta decisão é denominada **ligação dinâmica** (**dynamic binding**).

Considere o polimorfismo a seguir, *em métodos*:



Exemplo de polimorfismo II

Seguindo o exemplo, podemos observar o polimorfismo *nas variáveis*:

- Uma variável do tipo `Poligono` pode assumir a forma de `Poligono`, `Triangulo`, `Retangulo`, etc.

```
1 Poligono p1, p2, p3;
2
3 p1 = new Poligono();
4 ...
5 p2 = new Triangulo();
6 ...
7 p3 = new Retangulo();
```

Exemplo **sem** polimorfismo I

Área total de um *array* de polígonos, usando exemplo anterior, mas **sem sobrescrita de** `area()`:

```
1  double areaTotal() {
2      double areaTotal = 0;
3      for (int i = 0; i < MAXPOLIG; ++i) {
4          if (pol[i] instanceof Poligono)
5              areaTotal +=
6              pol[i].areaPoligono();
7          else if (pol[i] instanceof Triangulo)
8              areaTotal +=
9  ((Triangulo) pol[i]).areaTriangulo();
10         else if (pol[i] instanceof Retangulo)
11             areaTotal +=
12 ((Retangulo) pol[i]).areaRetangulo();
```

CAST!

```
13          else if (pol[i] instanceof Hexagono)
14              areaTotal +=
15  ((Hexagono) pol[i]).areaHexagono();
16          return areaTotal;
17      }
18  }
```

`instanceof`: palavra reservada para testar se objeto é de determinada classe, retornando `true` quando for o caso, e `false` caso contrário.

Exemplo **com** polimorfismo

Usamos polimorfismo de `area()` como no diagrama de classes visto:

```
1 double areaTotal() {
2     double areaTotal = 0;
3     for (int i = 0; i < MAXPOLIG; ++i) {
4         areaTotal += pol[i].area();
5     return areaTotal;
6 }
```

Rápido, enxuto e fácil de entender:

- O acréscimo de uma nova subclasse de `Poligono` **não altera nenhuma linha** do código acima.

Legibilidade do código:

- O mesmo nome para a mesma operação (método) facilita o aprendizado e melhora a legibilidade.

Código de menor tamanho:

- Código mais claro, enxuto e elegante.

Flexibilidade:

- Pode-se incluir novas classes sem alterar o código que a manipulará.

Classe abstrata I

Uma classe abstrata – ou **classe virtual** – é uma classe incompleta onde alguns ou todos os seus métodos não possuem implementação

Todas as classes vistas até este ponto não são abstratas, são *classes concretas*.

Classe abstrata II

Quando usamos herança, em diversas ocasiões as classes base são bastante *genéricas* (principalmente se houver *vários níveis de herança*);
Neste caso, pode-se implementar classes que definem *comportamentos genéricos* – as **classes abstratas**:

- A essência da superclasse é definida e pode ser *parcialmente* implementada;
- Detalhes são definidos em subclasses especializadas;
- **Não** podem ser instanciadas (servem apenas para reunir características comuns dos descendentes).

Java

Palavra reservada `abstract`

Classe abstrata III

Exemplo:

```
1 public abstract class Conta {
2     private int num;
3     private float saldo;
4 }
```

```
1 public class Poupanca extends Conta {
2     ...
3 }
```

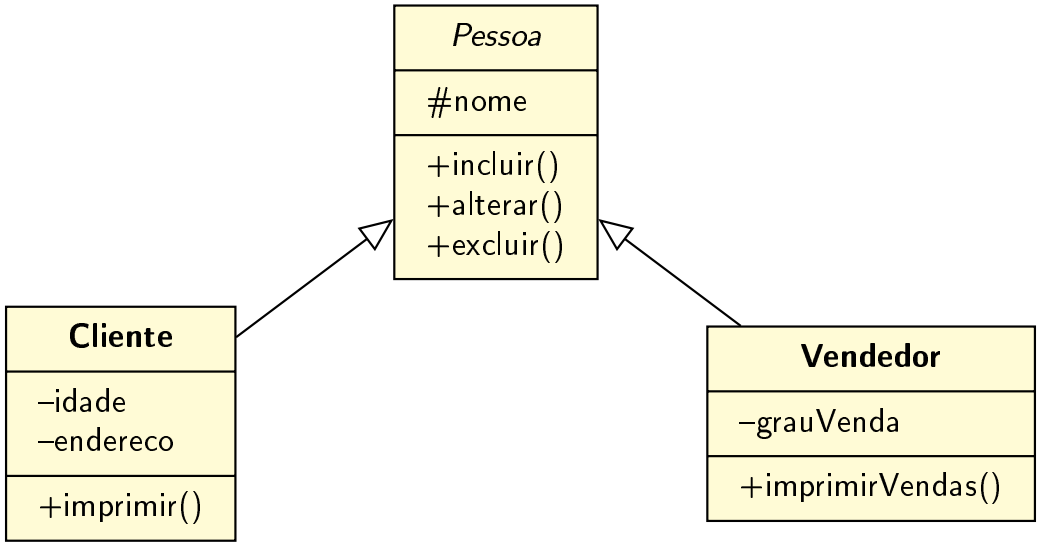
```
1 public class ContaEspecial extends Conta {
2     ...
3 }
```

Classe abstrata IV

```
1 Conta c; // esta linha está ok
2 c = new Conta(); // ERRO: não posso criar objeto
    de classe abstrata
3 c = new ContaEspecial(); // ok
```

Classe abstrata V

Outro exemplo (em UML, nome de classe abstrata é escrito em *itálico*):



- A classe `Pessoa` existe para reunir as características.
- Um objeto efetivo dentro de uma loja deve ser cliente ou vendedor. *Não existe apenas pessoa*.

Métodos abstratos são métodos definidos exclusivamente dentro de classes abstratas, mas *não são* implementados nas mesmas (apenas sua *assinatura* é especificada).

- Os métodos abstratos devem ser **obrigatoriamente** implementados em **toda classe herdeira (concreta)** da classe abstrata em que são definidos.

Declarar um método como abstrato é uma forma de obrigar o programador a redefinir esse método em todas as subclasses para as quais se deseja criar objetos.

Exemplo:

```
1 public abstract class Forma {
2     public abstract double perimetro();
3     public abstract double area();
4 }
```

```
1 public class Circulo extends Forma {
2     public double perimetro() {
3         return 2.0*Math.PI*this.raio;
4     }
5     public double area() {
6         return Math.PI*Math.pow(this.raio,2.0);
7     }
8 }
```

```java
public class Quadrado extends Forma {
    public double perimetro() {
        return 4.0*this.lado;
    }
    public double area() {
        return Math.pow(this.lado,2.0);
    }
}
```

- Crie um algoritmo para instanciar os objetos `BemTevi`, `Papagaio`, `Cachorro` e `Vaca`. Na prática, nunca iremos instanciar um `Animal`. A classe serve apenas para a definição de mamíferos e pássaros (subclasses). Da mesma forma, não instanciamos `Mamifero` nem `Passaro`. Somente instanciamos objetos `BemTevi`, `Papagaio`, `Cachorro` e `Vaca`. Logo, `Animal`, `Mamifero` e `Passaro` são classes abstratas.
- Crie um método abstrato em `Animal`, implementando-o nas classes concretas.

Referências

1. Addison-BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML, Guia do Usuário*. Rio de Janeiro: Campus, 2000.
2. FOWLER, M. *UML Essencial*, 2a Edição. Porto Alegre: Bookman, 2000.
3. LARMAN, C. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos*. Porto Alegre: Bookman, 2001.

Os slides dessa apresentação foram cedidos por:

- Graça Marietto e Francisco Zampirolli, UFABC
- Profa Katti Faceli, UFSCar/Sorocaba
- Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU

Como visto, o conjunto de métodos disponíveis em um objeto é chamado **interface**:
É através da interface que se interage com os objetos de uma determinada classe – através de seus métodos.

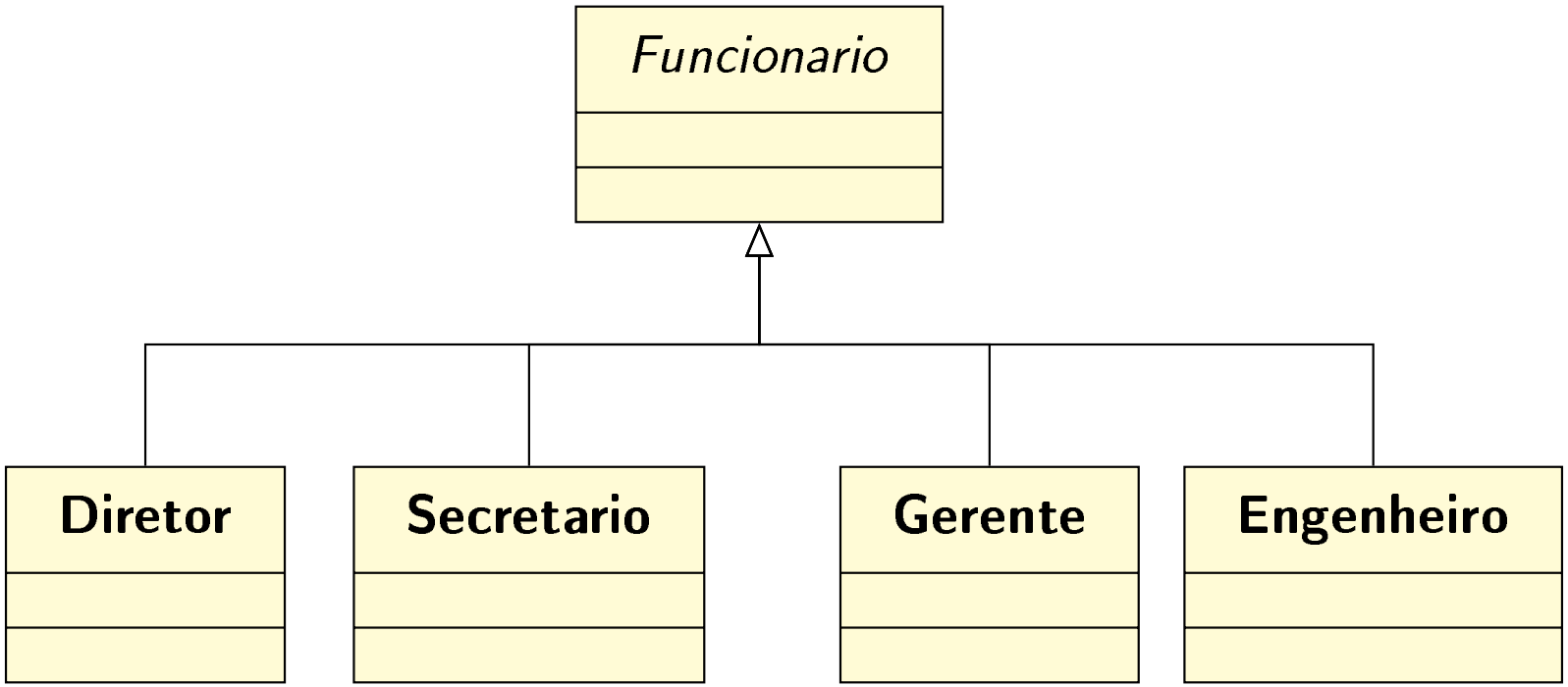
Uma definição mais formal:

Interface

"Contrato" assumido por uma classe, de modo a garantir certas funcionalidades a suas instâncias.

Em outras palavras, uma interface define *quais* métodos que uma classe **deve implementar** visando tais funcionalidades.

Considere o exemplo:



As classes acima definem o organograma de uma empresa ou banco, ao qual criamos um *sistema interno* que pode ser acessado *somente* por diretores e gerentes:

```
1 class sistemaInterno {
2     void login(Funcionario funcionario) {
3         funcionario.autentica(...); //  erro de semâ
              ntica: nem todo funcionario autentica.
4     }
5 }
```

Engenheiros e funcionários não autenticam no sistema interno.
Como resolver o problema, do ponto de vista de OO?

Motivação III

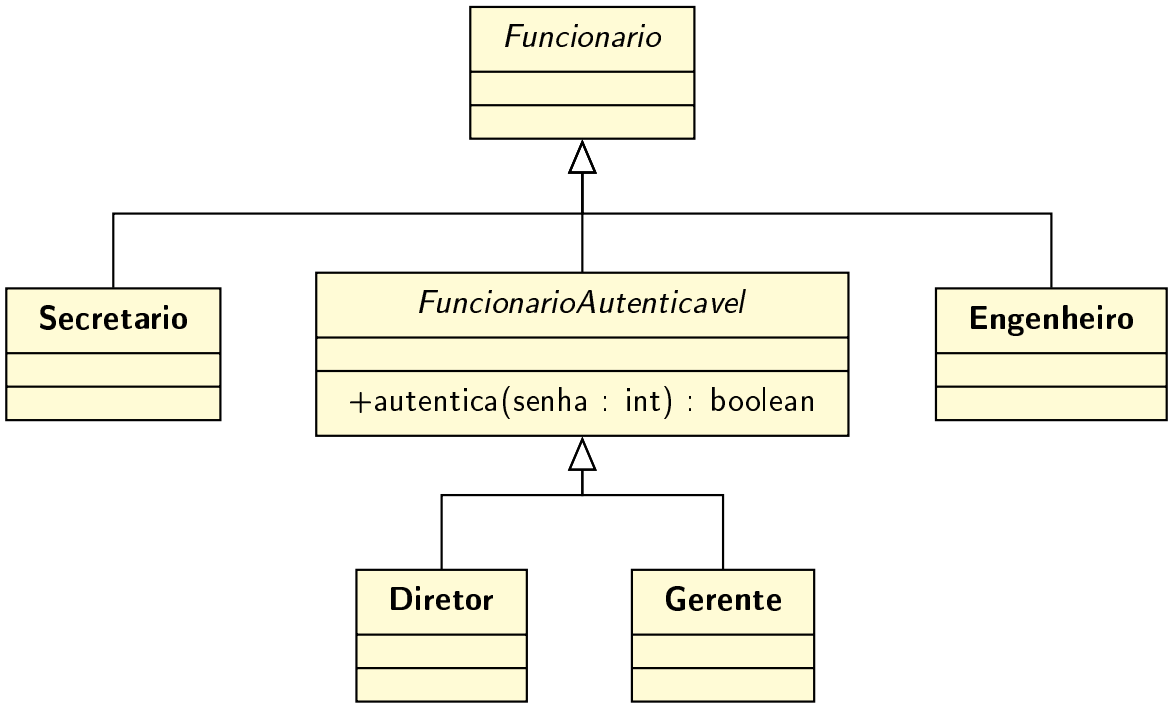
Alternativa (ruim):

```
1 class SistemaInterno {
2     void login(Diretor funcionario) {
3         funcionario.autentica(...);
4     }
5
6     void login(Gerente funcionario) {
7         funcionario.autentica(...);
8     }
9 }
```

Toda vez que outra classe que possa autenticar é definida, devemos adicionar novo método `login()`.

Motivação IV

Possível solução: subclasse intermediária, com o método `autentica()`.
Tal classe pode ou não ser abstrata:



Caso mais complexo: e se *cliente* da empresa ou banco também tiver direito a acessar o sistema interno, através de login?

`class Cliente extends FuncionarioAutenticavel { ... }` – **Herança sem sentido**!

Não há sentido `Cliente` herdar atributos e comportamentos de `Funcionario`, ex.: salário, bonificação, etc.

Herança é um tipo de relacionamento **"é um"**.

Solução: usar o conceito de **interface**.
O fato é que as classes `Gerente`, `Diretor` e `Cliente` possuem um fator comum, o método `autentica()` – porém apenas as duas primeiras correspondem a funcionários no sistema.

O "contrato", portanto, a ser assumido por tais classes, é que devem ser *autenticáveis* no sistema. Assim, cria-se a interface de nome, por exemplo, `Autenticavel`, possuindo a *assinatura* ou protótipo do método `autentica()`:

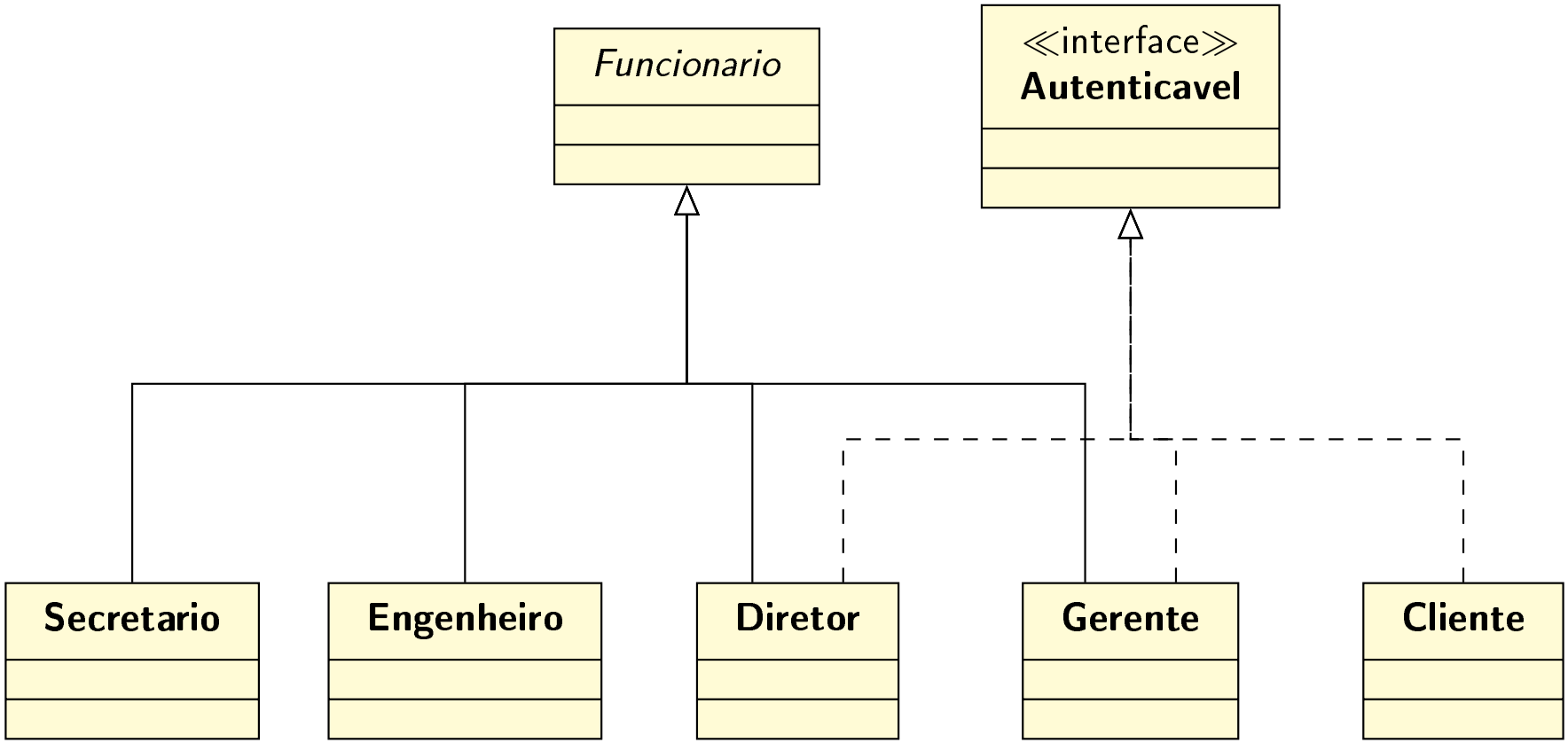
```
1 interface Autenticavel {
2     boolean autentica(int senha);
3 }
```

A interface diz **o que** o objeto deve fazer, mas não **como** fazer. Isto será definido na *implementação* da interface por uma classe.
Em Java: palavra reservada `implements`.

```
1 class Gerente extends Funcionario implements
      Autenticavel {
2     private int senha;
3     //...
4     public boolean autentica(int senha) {
5         // a implementação deste método é obrigató
              ria na classe Gerente.
6     }
7 }
```

```
1 class Cliente implements Autenticavel {
2     private int senha;
3     //...
4     public boolean autentica(int senha) {
5         // a implementação deste método é obrigató
              ria na classe Cliente.
6     }
7 }
```

Diagrama de classes:



O *polimorfismo* é uma vantagem ao se utilizar interfaces: podemos ter algo do tipo

`Autenticavel a = new Gerente();`

A variável do tipo `Autenticavel` pode se referir a objeto de *qualquer* classe que implemente `Autenticavel`. E o sistema interno visto fica como:

```
1 class SistemaInterno {
2     void login(Autenticavel a) {
3         // lê uma senha ...
4         boolean ok = a.autentica(senha);
5         // objeto que autentica não é mais necessariamente
              funcionário...
6     }
7 }
```

Interface VI

Usos da interface I

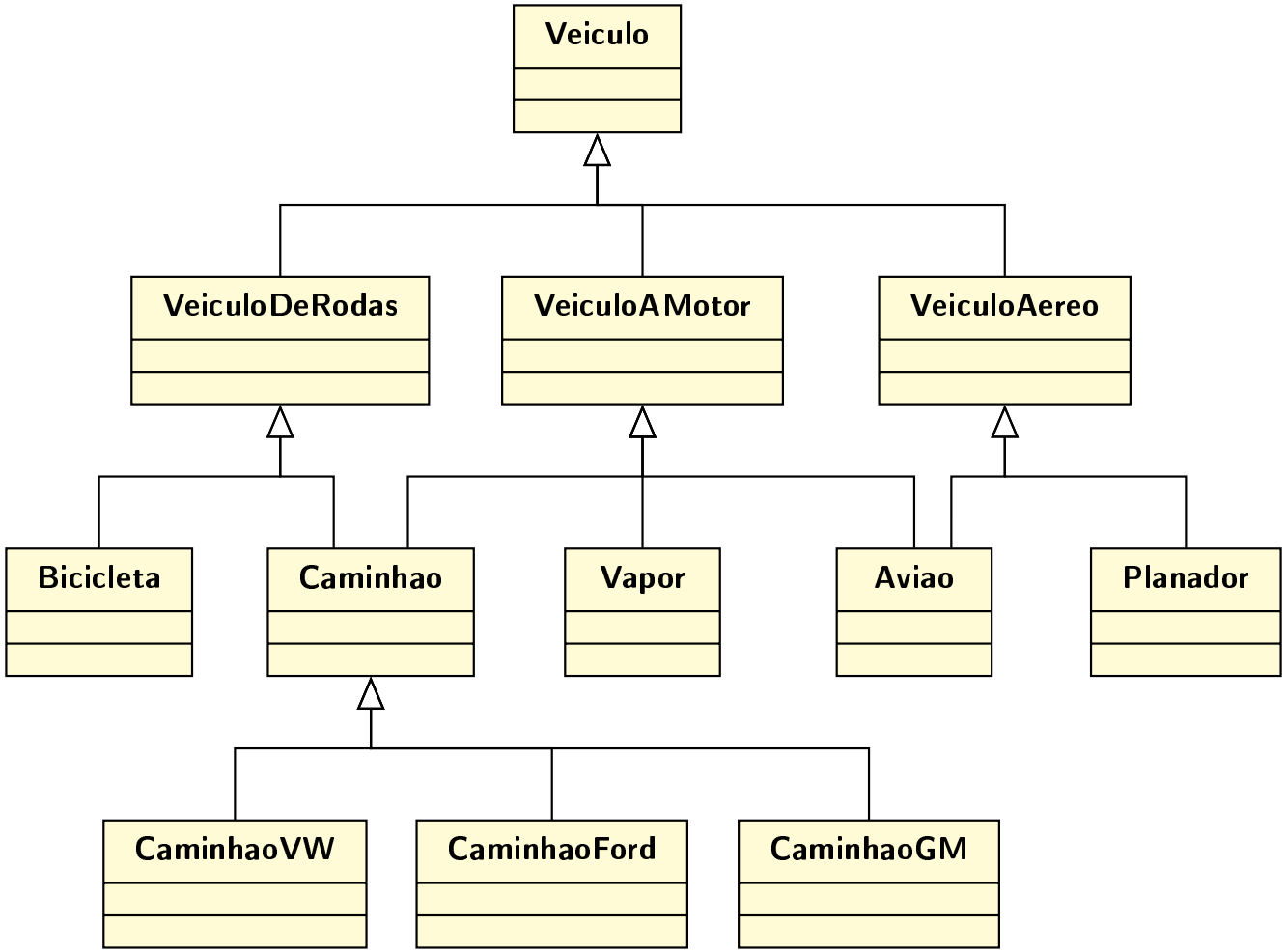
Vimos dois usos para interface:

- Capturar *similaridades de comportamento* entre classes não relacionadas diretamente (definir superclasse abstrata não seria natural, pois não há "parentesco" forte entre as classes);
- Declarar métodos que uma ou mais classes não relacionadas *devem* necessariamente implementar.

Um outro uso:

- Revelar uma *interface de programação*, sem revelar a classe que a implementa (apresenta a interface pública, mas não revela onde está a implementação).

Herança múltipla I



Em Java – e em muitas linguagens OO, a **múltipla herança** – em que uma classe possui mais de uma superclasse – não é permitida nativamente. Isto é, a instrução abaixo **não** é possível:

```
public class Aviao extends VeiculoAMotor, VeiculoAereo {...}
```

A restrição evita conflitos que podem ocorrer devido aos *atributos* herdados das superclasses, que podem se sobrepor, afetando a definição do *estado* de um objeto da subclasse.
Mais detalhes em https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html

Interfaces:

- Não possuem atributos;
- Não possuem construtores;
- Contém **assinaturas de métodos**:
  - Métodos em interfaces são **abstratos** (e **públicos**).

Como não temos atributos sendo definidos nas interfaces, eliminamos a restrição vista anteriormente:
É possível que uma classe implemente várias interfaces (ideia de herança múltipla).

Herança múltipla IV

```java
public interface VeiculoAMotor {
    void ligarMotor();
}
```

```java
public interface VeiculoAereo {
    void voar();
}
```

Herança múltipla V

```
1  public class Aviao implements VeiculoAMotor, VeiculoAereo {
2      // atributos...
3
4      public void ligarMotor() {
5          // classe Aviao deve implementar este método
6      }
7
8      public void voar() {
9          // classe Aviao deve implementar este método
10     }
11
12     ...
13 }
```

Uma interface também pode herdar características de outra(s) interface(s) – e *somente* de interface(s) – via herança:

```
public interface Interface4 extends Interface1, Interface2,
Interface3 { ...
```

Outras observações

- Interfaces são codificadas em arquivo próprio, com mesmo nome da interface;
- Classes que usam uma interface têm que implementar **todos os métodos** definidos na interface – mas há exceções (a partir Java 8);
- Uma classe pode implementar mais de uma interface, desde que não haja **conflitos de nomes**.

Conflitos de nomes e herança múltipla I

```
1 interface A {
2     tipo metodo(pars);
3 }
```

```
1 interface B {
2     tipo metodo(pars);
3 }
```

Se uma classe implementa essas duas interfaces, haverá conflito de nomes.

Conflitos:

- Se os métodos têm o **mesmo nome**, mas parâmetros diferentes, não há conflito, há **sobrecarga**;
- Se os métodos tiverem a **mesma assinatura**, a classe implementará *um método apenas*;
- Se as assinaturas dos métodos diferirem **apenas** no tipo de retorno, a classe não poderá implementar as duas interfaces – este na verdade é o único conflito não-tratável.

Supor uma classe `Classificador` com um método `ordena()`, que faz a ordenação de objetos de outras classes.

O método `ordena()` implementa um algoritmo de ordenação que compara todos os elementos usando o método `eMaiorQue()`;

Toda classe que quiser ter ordenação de seus objetos **deve implementar** o método `eMaiorQue()`.

Mais exemplos II

```
1 public class Classificador {
2     void ordena(Object[] a) {
3         if (a[i].eMaiorQue(a[i+1])) {
4             ...
5         }
6     }
7 }
```

Como garantir que toda classe que necessite de ordenação implemente o método `eMaiorQue()`? **Usar interfaces.**

Mais exemplos III

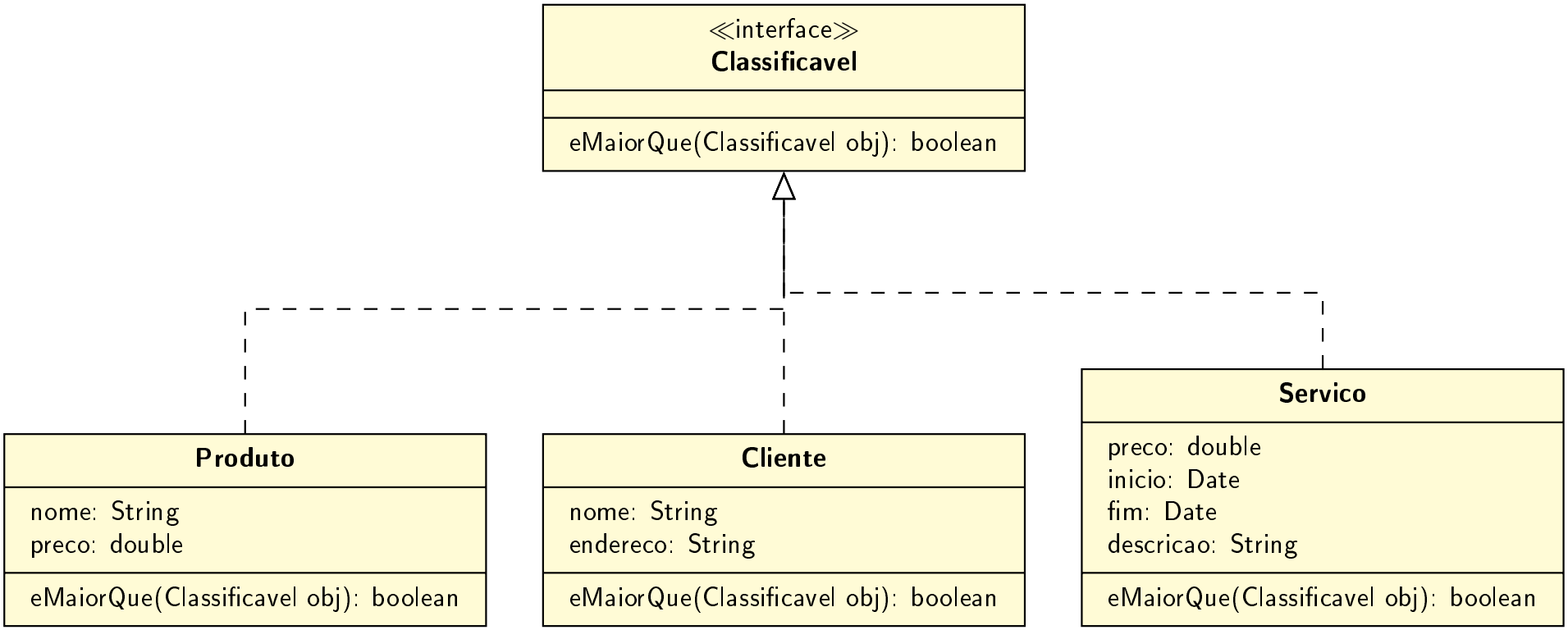
```
1 public interface Classificavel {
2     boolean eMaiorQue(Classificavel obj);
3 }
```

```
1 public class Produto implements Classificavel {
2     String nome;
3     double preco;
4     ...
5     boolean eMaiorQue (Classificavel obj)
6     { ... }
7     ...
8 }
```

```
1 public class Cliente implements Classificavel {
2     String nome;
3     String endereco;
4     ...
5     boolean eMaiorQue (Classificavel obj)
6     { ... }
7     ...
8 }
```

```java
public class Servico implements Classificavel {
    double preco;
    Date inicio, fim;
    String descricao;
    ...
    boolean eMaiorQue (Classificavel obj)
    { ... }
    ...
}
```

Mais exemplos VII



Foi visto que interfaces não contêm atributos, e sim **assinaturas de métodos** (isto é, métodos abstratos).
Além de tais membros, as interfaces também podem conter *constantes*; e métodos *estáticos* e *default*[1] (implementação padrão)

Constantes em interfaces:

Implicitamente, são `public`, `static` (i.e., da classe) e `final` (não é necessário escrever tais modificadores).

- `final`: indica que não pode ser alterado (equivalente a `const` em C)

```
1 public interface A {
2     // base of natural logarithms
3     double E = 2.718282;
4
5     // method signatures
6     void doSomething (int i, double x);
7     int doSomethingElse(String s);
8 }
```

E: valor *constante* – note que o valor é atribuído logo na declaração.

Por extensão, **enumerações** (listas de constantes) também são permitidos:

```
1 public interface B {
2     enum diaSemana {
3         DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA
4     }
5
6     // assinaturas de métodos...
7
8 }
```

Constantes em Java, por convenção, sempre em **caixa alta** (maiúsculas) – veja http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html. Mais de uma palavra: separam-se com _ (exemplo: `SEGUNDA_FEIRA`).

Antes da versão 8 do Java:
Toda vez que se quisesse atualizar a especificação de uma interface, adicionando-se a mesma um novo método, **todas as classes** que previamente a implementavam deveriam também implementar tal método, ou não compilariam.

Uma opção: colocar o novo método numa nova interface, **sub-interface** da anterior:

```
1 public interface B extends A {
2     tipo novoMetodo (...);
3 }
```

(uso agora opcional: quem quiser usar nova interface implementa `B`, mas quem quiser manter requisitos anteriores continua implementando `A`).

Java 8 – Processo simplificado com novo recurso:
Métodos *default*: a implementação *padrão* **é feita na interface** (não são abstratos); (re-)implementação nas classes torna-se *facultativa* –
Classes que implementam uma interface contendo um ou mais métodos *default* pode, mas não precisa implementá-los – já os terá definidos.

```
1 public interface A {
2     // outros métodos...
3
4     default tipo novoMetodo (...) {
5         // implementação aqui
6     }
7 }
```

Agora, `novoMetodo` pode ser chamado para qualquer objeto de classe que implemente `A`.

Herança entre interfaces e métodos *default*:
Se `B` é sub-interface de `A`, que define método *default* `novoMetodo`:

- B não menciona tal método, apenas o herdando como método *default*;
- **Redeclara** o método, **tornando-o abstrato**:
  - Toda classe que implementar `B`, e não `A`, precisará implementar `novoMetodo`;
- **Redefine** o método, reimplementando-o (será *default* também em `B`):
  - Toda classe que implementar `B`, e não `A`, terá como implementação padrão (*default*) de `novoMetodo` a dada em `B` – e não em `A`.

Também em Java 8: **Métodos estáticos**

- Associados a classe/interface que os define, e não a objetos.
- Não podem ser redefinidos via polimorfismo dinâmico.

```
1 public interface A {
2     static double sqrt2() {
3         return Math.sqrt(2.0);
4     }
5 }
```

Outros membros da interface VIII

```
1 public class Classe implements A {
2     // ...
3     public void mostraSqrt2() {
4         System.out.println("Raiz de 2 = " + A.sqrt2()); // sqrt2 é
              estático, dado pela interface A
5     }
6 }
```

[1] Java 8

Classe abstrata	**Interface**
Uma classe pode ser subclasse de apenas uma classe abstrata	Uma classe pode implementar múltiplas interfaces
Faz parte de uma hierarquia que possui correlação (“é-um”)	Não faz parte de uma hierarquia (classes sem relação podem implementar uma mesma interface)

Em Java é possível armazenar um conjunto de valores, primitivos ou objetos, utilizando variáveis compostas homogêneas (vetores, matrizes, etc)

Mas e se quisermos:

- Criar estruturas que aloquem dinamicamente espaço em memória (aumentar ou diminuir o espaço **em tempo de execução**)?
- Criar estruturas de dados mais complexas com disciplinas de acesso, através da implementação de tipos abstratos de dados como **listas**, **pilhas** e **filas**?

Estas questões são tratadas comumente em disciplinas específicas de **estruturas de dados**.

Na linguagem de programação Java, estas estruturas são oferecidas através do **Java Collections Framework**.

Arrays são estruturas de dados poderosas, mas com utilização específica:

- Inadequados para excluir/incluir elementos frequentemente.

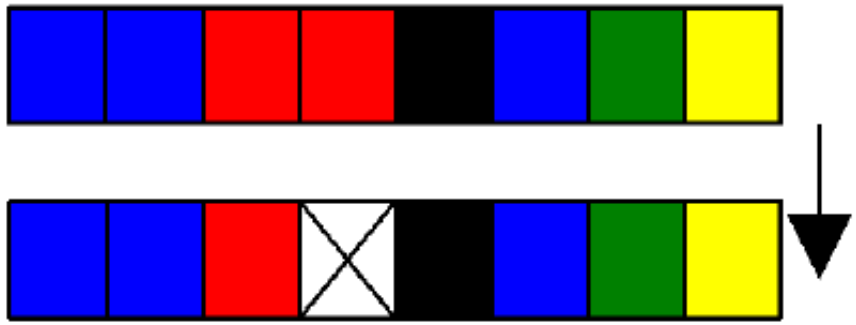
Em geral, não existe uma estrutura de dados que tenha desempenho excelente para várias operações que se possa realizar, como:

- Incluir, excluir, alterar, listar, ordenar, pesquisar, etc.

Além disso, manipular os *arrays* do Java é bastante trabalhoso:

- Não se pode redimensionar;
- A busca direta por um determinado elemento cujo índice não se conhece não é possível;
- Não se sabe quantas posições do *array* foram efetivamente usadas, sem uso de recursos auxiliares, como *contadores*.

Exemplo: remoção em posição intermediária de um *array*.



- Ao inserir novo elemento: procurar posição vazia?
- Armazenar lista de posições vazias?
- E se não houver espaço vazio? (`arraycopy()` não é bom)
- E qual o tamanho da estrutura? (posições de fato usadas?)

Um outro ponto importante:
ordenação.

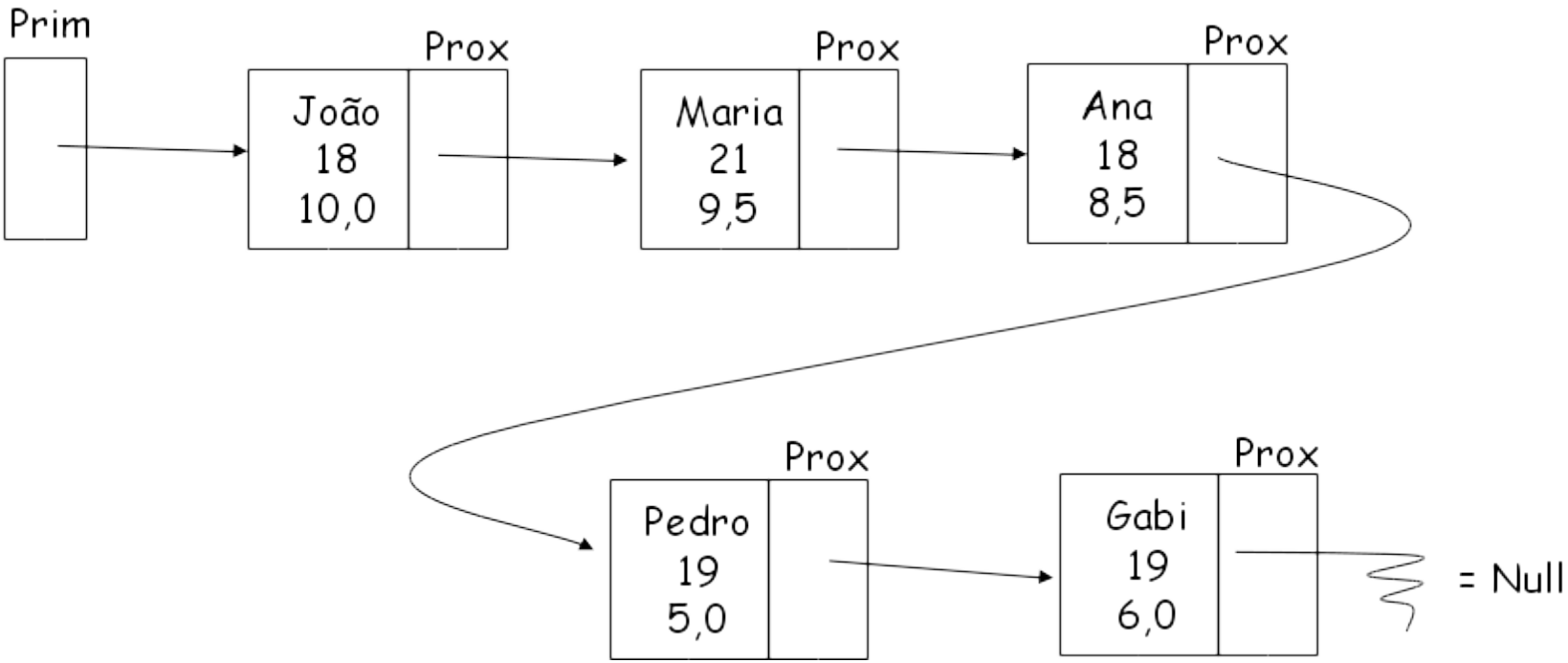
O que existem são estruturas de dados que são **"melhores"** para algumas operações:
A decisão depende do problema específico.

Algumas estruturas de dados:

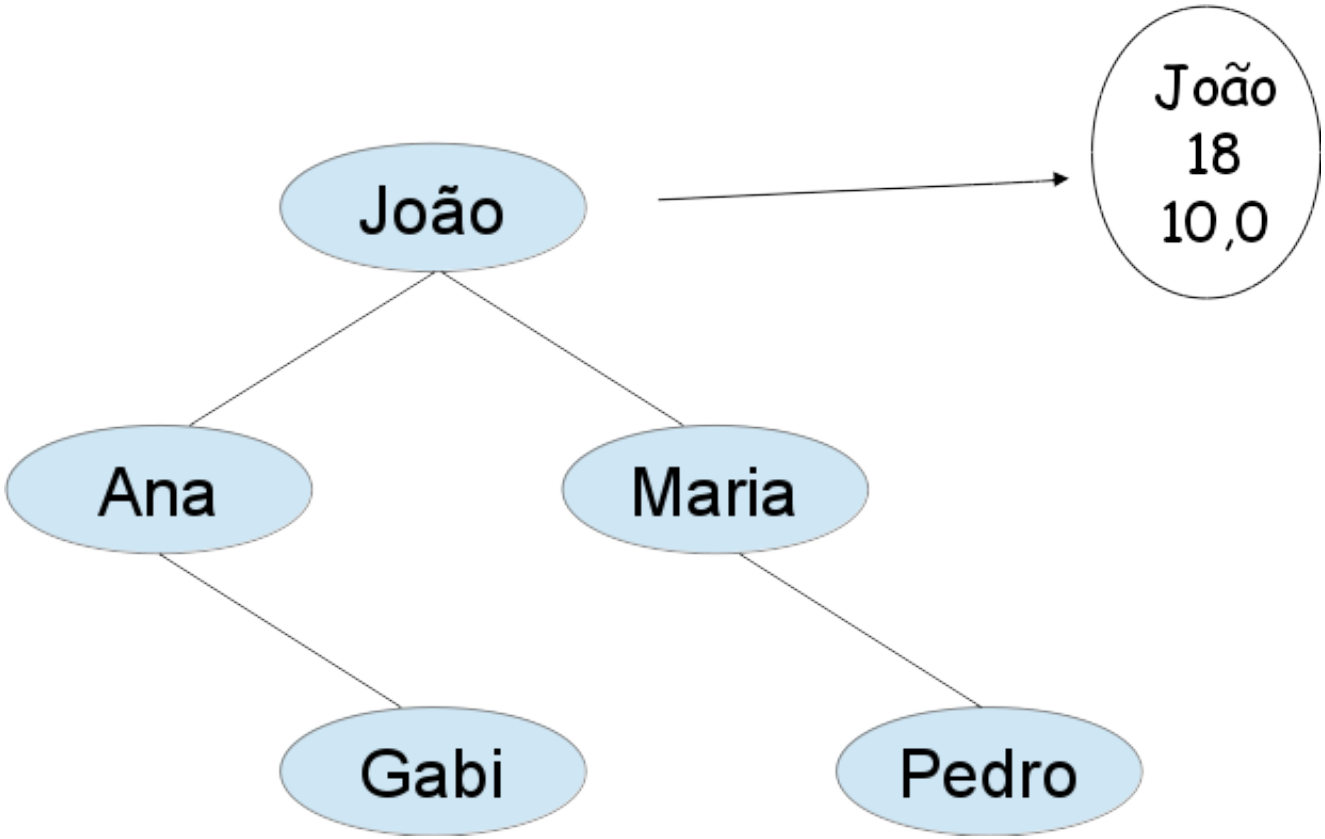
- Vetores;
- Listas encadeadas;
- Pilhas;
- Árvores;
- Tabelas *hash*;
- Etc.

Exemplo de *array*

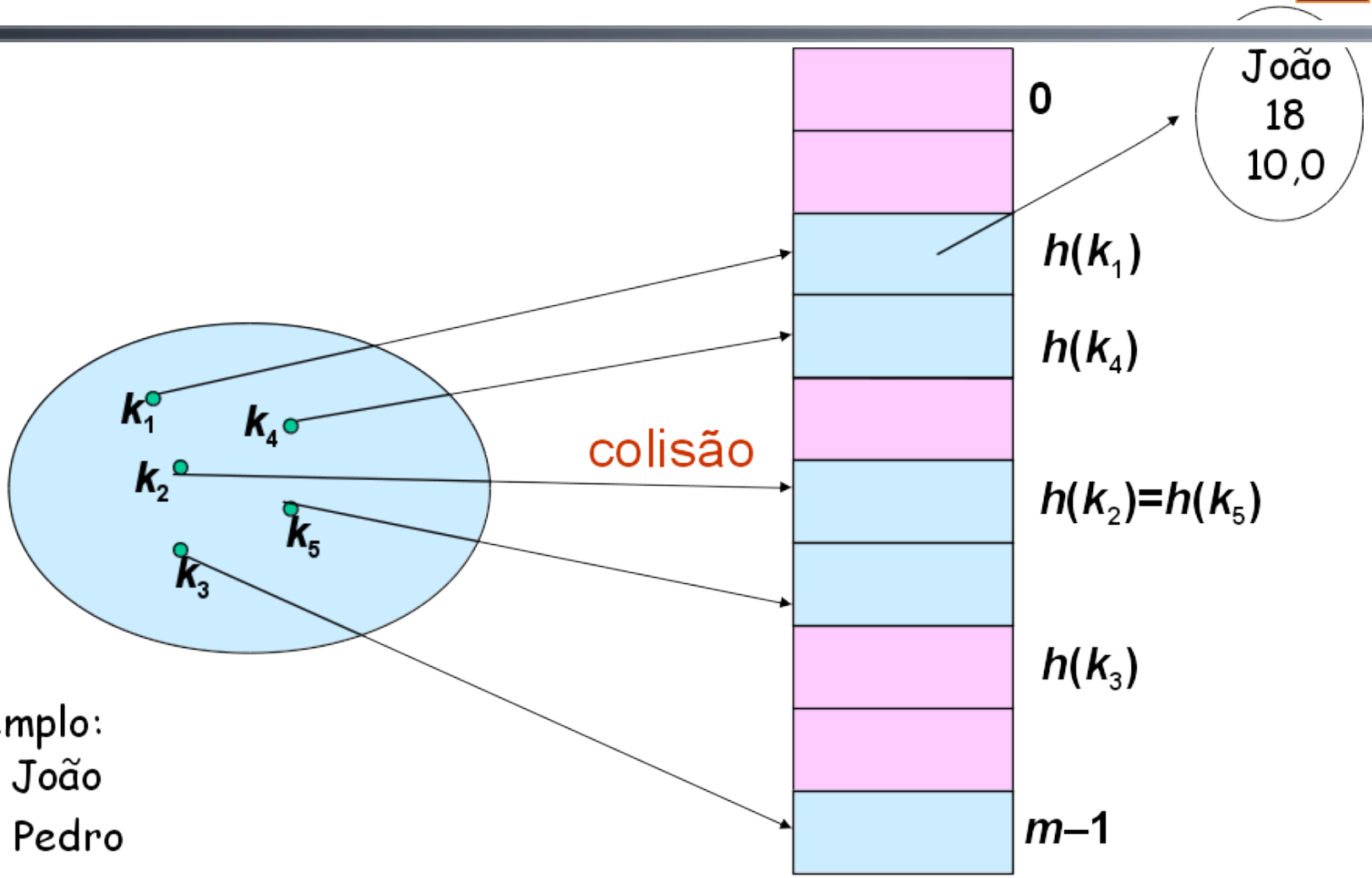
Exemplo de lista (encadeada)



Exemplo de árvore (árvore binária de busca)



Exemplo de tabela *hash*



Exemplo:

k_1 = João

k_2 = Pedro

k_3 = Maria

k_4 = Ana

k_5 = José

Como podemos ver, existem ED especializadas em certas operações/funcionalidades.

E em Java?

A partir do Java 1.2 (Java 2):
Collections framework

Collections framework

Conjunto de **classes** e **interfaces** Java, dentro do pacote nativo `java.util`, que representam diversas estruturas de dados avançadas.

- em outras palavras, são implementações pré-existentes em Java para EDs bem conhecidas;
- Possuem métodos para *armazenar*, *recuperar*, *consultar*, *listar* e *alterar* dados que são tratados de forma agregada;

Uma definição para o que seria uma *coleção*:

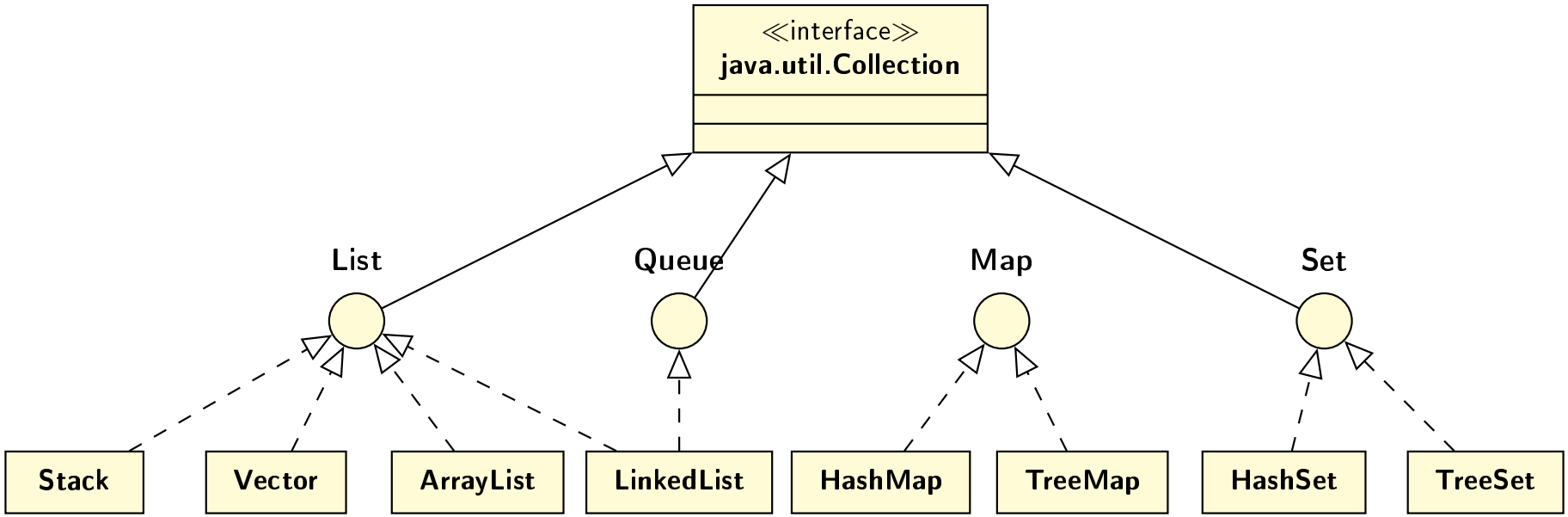
Coleção

Um *objeto* que agrupa múltiplos elementos em uma estrutura única

Exemplos de situações em que se usa coleções:

- Lista de disciplinas, lista de professores, lista de alunos, lista de turmas, lista de alunos por turma, etc;
- Relação de temperaturas atmosféricas de uma localidade para um determinado período;
- Conjuntos de dados que não apresentam elementos repetidos, como clientes que receberam presente de Natal (podem vir de listas de diferentes vendedores);
- Filas (por exemplo, clinica médica ou supermercado), onde o primeiro a chegar é o primeiro a ser atendido.

Diagrama UML



Incompleto: para estrutura completa, visitar: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Core Collections: principais coleções

- `Set` (java.util.Set)
- `List` (java.util.List)
- `Queue` (java.util.Queue)
- `Map` (java.util.Map)

Não oferecem nenhuma implementação diretamente: são **interfaces**.

Interface Set (define uma coleção que não contém objetos duplicados). Como o nome sugere, modela a abstração de *conjunto* (matemática)

- `HashSet` é a implementação mais comum.

Interface List (define uma sequência de objetos onde é possível elementos duplicados; inserção qualquer lugar).

- `ArrayList` é a implementação mais comum.

Interface Map (define uma coleção em um algoritmo *hash*).

- `HashMap` é a implementação mais comum.

Interface Queue (define uma coleção que representa uma fila, ou seja, implementa o modelo FIFO – *first-in-first-out*).

- `LinkedList` é a implementação mais comum.

Classes que implementam `java.util.List`

- `ArrayList`: implementação de lista usando para armazenar os dados um *array* redimensionável. Melhor performance para métodos get (acesso a elemento) e `set` (alterar);
- `LinkedList`: lista duplamente encadeada (melhor performance p/ métodos `add` e `remove` – inserção e remoção);
- `Stack`: pilha (estrutura LIFO: *last-in-first-out*);
- `Vector`: `ArrayList`, melhorado para trabalhar com **código paralelo**.

Criação de um `ArrayList` de objetos de uma classe chamada `Aluno`:

```
1 ArrayList<Aluno> alunos;
2 alunos = new ArrayList<Aluno>();
3
4 Aluno a = new Aluno();
5 alunos.add(a);
```

Utilização de um ArrayList na classe Turma:

```
1 public class Turma {
2     private  List<Aluno>  alunos;
3     Turma {
4         alunos = new ArrayList<Aluno>();
5     }
6 }
```

Outro exemplo:

```
1  import java.util.ArrayList;
2
3  public class Cores {
4      public static void criarCores() {
5          ArrayList<String> cores = new ArrayList<>();
6          cores.add("Vermelho");
7          cores.add("Verde");
8          cores.add("Azul");
9          cores.add("Amarelo");
10         for (int i = 0; i < cores.size(); i++) {
11             String str = cores.get(i);
12             System.out.println(str);
13         }
14         cores.remove(3);
15             cores.remove("Azul");
16         System.out.println("==================");
```

```
17          for (String s : cores) {
18              System.out.println(s);
19          }
20          int indice = cores.indexOf("Vermelho");
21          cores.set(indice, "Preto");
22          System.out.println("========================");
23          for (String s : cores) {
24              System.out.println(s);
25          }
26      } //Fim do método criarCores()
27
28      public static void main(String args[]) {
29          criarCores();
30      }
31  }
```

Ordenação: `Collections.sort()` I

O *framework* de coleções do Java possui uma classe, também do pacote `java.util`, chamada `Collections` – não confundir com interface `Collection` vista previamente – que oferece, dentre outros métodos, um **método de ordenação**, o método `sort()`.
Basta importar `Collections`, na classe onde irá utilizar a ordenação,

```
import java.util.Collections
```

Ordenação: Collections.sort() II

Exemplo com *strings*:

```
1 List<String> lista = new ArrayList<>();
2 lista.add("verde");
3 lista.add("azul");
4 lista.add("preto");
5
6 System.out.println(lista);
7 Collections.sort(lista);
8 System.out.println(lista);
```

Ordenação: `Collections.sort()` III

No exemplo anterior, `ArrayList` de *strings* foi ordenado.

E se trabalharmos com objetos de outra classe? Como fica a ordenação?

Ordenação: Collections.sort() IV

Exemplo:

```
1  ContaCorrente c1 = new ContaCorrente();
2  c1.deposita(500);
3  ContaCorrente c2 = new ContaCorrente();
4  c2.deposita(200);
5  ContaCorrente c3 = new ContaCorrente();
6  c3.deposita(150);
7
8  List<ContaCorrente> contas = new ArrayList<>();
9  contas.add(c1);
10 contas.add(c3);
11 contas.add(c2);
12
13 Collections.sort(contas); // qual seria o critério para esta
       ordenação?
```

Ordenação: `Collections.sort()` V

Considere que a classe `ContaCorrente` possui um atributo chamado `saldo` e um método chamado `deposita()`, que altera o valor do saldo.

Neste caso, é preciso **instruir** o método `sort()` sobre como será o critério de ordenação, ou seja, como os elementos serão **comparados**.

Isto será feito *implementando-se a interface* `Comparable` do pacote `java.lang`.

Ordenação: `Collections.sort()` VI

A interface `Comparable` possui um **método abstrato** chamado `compareTo()`, que compara um objeto qualquer em relação a outro, e *retorna um inteiro* de acordo com a comparação:

- < 0, se o objeto que chama o método (`this`) é "menor que" o objeto passado por parâmetro do método;
- 0, se ambos são iguais;
- > 0, se `this` é "maior que" o objeto passado.

O método `sort()` de `Collections` chamará o método `compareTo()` internamente.

Ordenação: Collections.sort() VII

```
1  public class ContaCorrente implements Comparable<
       ContaCorrente> {
2      private double saldo;
3
4      // ... demais atributos, e outros métodos ...
5
6      public int compareTo(ContaCorrente outra) {
7          if (this.saldo < outra.saldo) {
8              return -1;
9          } else if (this.saldo > outra.saldo) {
10             return 1;
11         } else {
12             return 0; // saldos iguais
13         }
14     }
15 }
```

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

Ordenação: `Collections.sort()` VIII

Outros métodos de `java.lang.Collections`

- `binarySearch(List, Object)`: Realiza uma **busca binária** por determinado elemento na lista ordenada e retorna sua posição ou um número negativo, caso não encontrado.
- `max(Collection)`: Retorna o maior elemento da coleção.
- `min(Collection)`: Retorna o menor elemento da coleção.
- `reverse(List)`: Inverte a lista.

Outros métodos, e mais detalhes sobre a classe `Collections`: https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

EDs e respectivas classes

Interface	Tabela *hash*	Lista estática (*array*)	Árvore balanceada	Lista encadeada	Tabela *hash* com lista encadeada
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`		`ArrayDeque`		`LinkedList`	
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

1. Apostila de Java e POO Caelum: disponível em https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf – acesso em: MAI/2017.
2. Documentação Java Oracle: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html, https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html

Os slides de parte desta seção foram cedidos por Marcelo Z. do Nascimento, FACOM/UFU
LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU

No tratamento de problemas usando OO (e mesmo outros paradigmas de programação), precisamos obedecer certas **regras**, ou **restrições** que fazem parte do problema

No caso de OO, podemos mencionar, por exemplo, *regras de negócio* a serem respeitadas na implementação dos métodos
Exemplo: `cal.defineDiaDoMes(35);` – data inválida: método deve evitar atribuição do valor 35 como dia do mês do objeto `cal`.

Como avisar quem chamou o método de que não foi possível realizar determinada operação?

Uma opção natural seria usar o **retorno** do método, porém há problemas...

Imagine a situação a seguir:

```
1  public Cliente procuraCliente(int id) {
2      if (idInvalido) {
3          // avisa o método que chamou este que
               ocorreu um erro
4      } else {
5          Cliente cliente = new Cliente();
6          cliente.setId(id);
7          // cliente.setNome("nome do cliente");
8          return cliente;
9      }
10 }
```

Não é possível descobrir se houve erro através do retorno, pois método retorna um objeto da classe `Cliente`.

Outro caso:

```
1 Conta minhaConta = new Conta();
2 minhaConta.deposita(100);
3 // ...
4 double valor = 5000; // valor acima do saldo (100)
      + limite
5 minhaConta.saca(valor); // saca: booleano. Vai
    retornar false, mas ninguém verifica!
6 caixaEletronico.emite(valor);
```

Nos casos anteriores, podemos ter uma situação onde o id do cliente é inválido, ou o valor para saque é muito alto ou mesmo inválido (ex. negativo), o que configuram **exceções** a tais regras de negócio.

Exceção

Algo que normalmente não ocorre – não deveria ocorrer – indicando algo estranho ou inesperado no sistema

Quando uma exceção é *lançada* (***throw***):

- JVM verifica se método em execução toma precaução para tentar contornar situação:
  - Se precaução não é tomada, a execução do método **para** e o teste é repetido no método anterior – que chamou o método problemático.
  - Se o método que chamou não toma precaução, o processo é repetido até o método `main()`;
  - Por fim, se `main()` não trata o erro, a JVM "morre".

Tratando exceções: mecanismo ***try*** / ***catch***.

- **Try**: *tentativa* de executar trecho onde pode ocorrer problema;
- **Catch**: quando exceção lançada em tal trecho, a mesma é *capturada*. Seu tratamento é feito no bloco do |catch|.

Exemplo – sem *try/catch*

```
1 class TesteErro {
2     public static void main(String[] args) {
3         int[] array = new int[10];
4         for (int i = 0; i <= 15; i++) {
5             array[i] = i;
6             System.out.println(i);
7         }
8     }
9 }
```

Saída:

```
0
1
2
3
4
5
6
7
8
9
Exception in thread "main" java.lang.
    ArrayIndexOutOfBoundsException: 10
at excecoes.TesteErro.main(TesteErro.java:5)
```

Adicionando *try/catch* no código anterior:

```
1  class TesteErro {
2      public static void main(String[] args) {
3          int[] array = new int[10];
4          for (int i = 0; i <= 15; i++) {
5              try {
6                  array[i] = i;
7                  System.out.println(i);
8              } catch (ArrayIndexOutOfBoundsException e) {
9                  System.out.println("Erro: " + e);
10             }
11         }
12     }
13 }
```

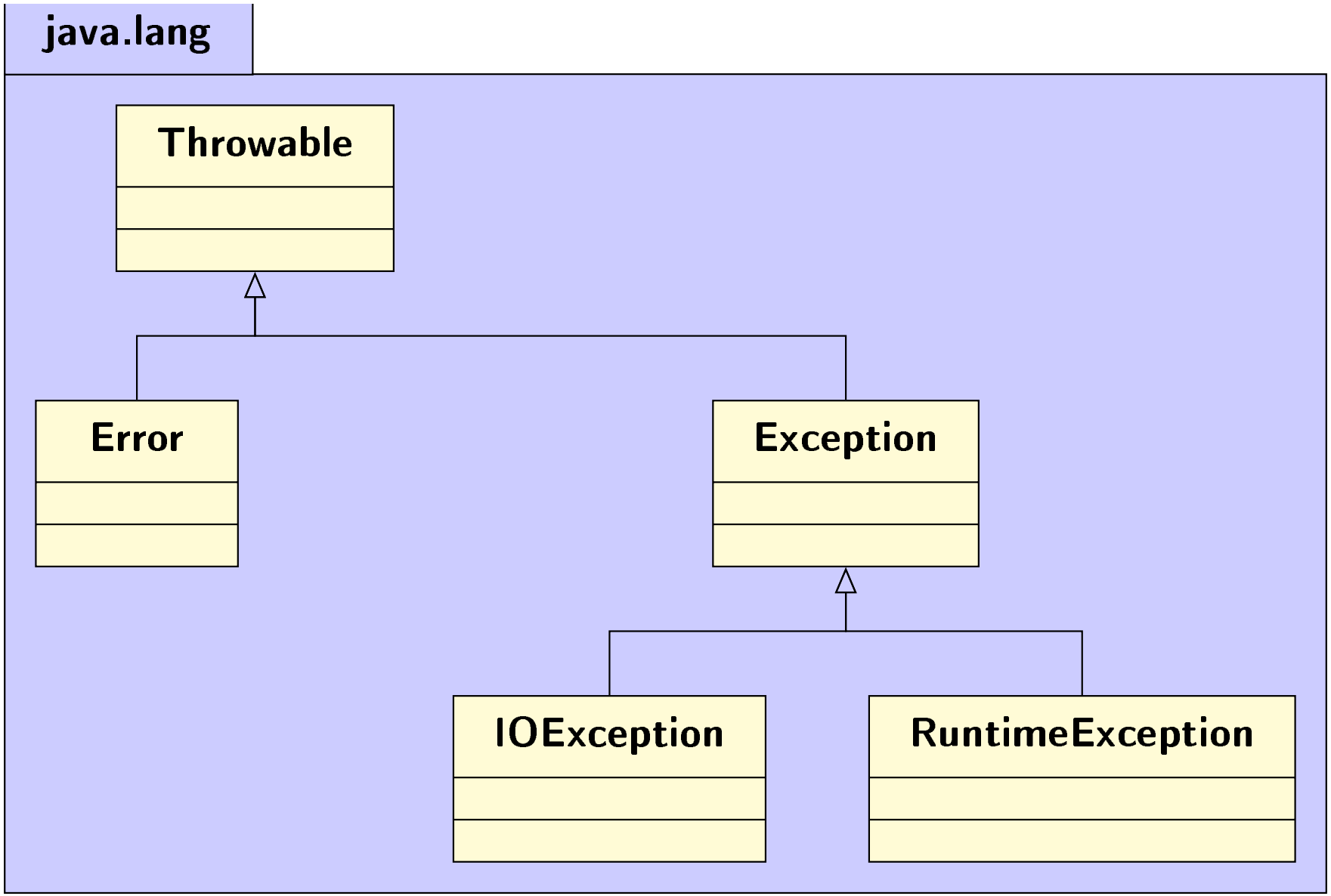
Saída:

```
...
7
8
9
Erro: java.lang.ArrayIndexOutOfBoundsException: 10
Erro: java.lang.ArrayIndexOutOfBoundsException: 11
Erro: java.lang.ArrayIndexOutOfBoundsException: 12
Erro: java.lang.ArrayIndexOutOfBoundsException: 13
Erro: java.lang.ArrayIndexOutOfBoundsException: 14
Erro: java.lang.ArrayIndexOutOfBoundsException: 15
```

Note que quando a exceção é capturada, a execução do método `main()` procede normalmente, até o encerramento do programa.

A exceção `ArrayIndexOutOfBoundsException` é, na verdade, uma classe contida dentro do pacote java.lang.

Família Throwable I



A hierarquia segue além: `ArrayIndexOutOfBoundsException`, por exemplo, é subclasse de `RuntimeException`. Outros exemplos de subclasses incluem:

- `ArithmeticException`. Ocorre, por exemplo, quando se faz divisão por 0;
- `NullPointerException`. Referência nula.

A classe `Error` define um tipo de erro causado pelo sistema, ex: `StackOverflowError`, `OutOfMemoryError`, e **não pode ser lançado** diretamente pelo programador;
Mais detalhes: `https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html`

A hierarquia `Exception`, por sua vez, se divide em **vários ramos**.
`RuntimeException` são os erros em tempo de execução, de **captura não obrigatória** (*unchecked exception* ou **exceção não-verificada**);
e `IOException` são apenas dois exemplos.
Todas as exceções que não são `RuntimeException` ou suas subclasses devem ser capturadas e tratadas (*checked exceptions* ou **exceções verificadas**).
Mais detalhes: `https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html`

Em caso de não tratamento das exceções *checked*, o compilador acusará erro, impedindo a geração do *bytecode*.
Exemplo:

```
1 class Excecoes {
2     public static void main(String[] args) {
3         new java.io.FileInputStream("arquivo.txt");
4     }
5 }
```

Pode ocorrer `FileNotFoundException` (subclasse de `IOException`).
NetBeans sugere duas formas de tratamento, que veremos a seguir.

Quando uma exceção é lançada, há duas formas de tratamento, do ponto de vista de um **método**:

- Envolvê-la em um *try-catch* (*surround with try-catch*), como já foi visto.
- Delegar o tratamento da exceção para o método que chamou o atual (*add throws clause*).

Try-catch:

```
1  import java.io.FileNotFoundException;
2
3  class Excecoes {
4      public static void main(String[] args) {
5          try {
6              new java.io.FileInputStream("arquivo.txt");
7          } catch (FileNotFoundException ex) {
8              System.out.println("Não foi possível abrir o
                   arquivo para leitura.");
9          }
10     }
11 }
```

Passando o tratamento para o método que invocou o atual (cláusula `throws`):

```
1 import java.io.FileNotFoundException;
2 class Excecoes {
3     public static void main(String[] args) throws
          FileNotFoundException {
4         new java.io.FileInputStream("arquivo.txt");
5     }
6 }
```

É desnecessário tratar no `throws` as *unchecked exceptions*, porém é permitido, e pode facilitar a leitura e documentação do código.

A propagação pode ser feita através de vários métodos:

```
1  public void teste1() throws FileNotFoundException {
2      FileReader stream = new FileReader("c:\\teste.txt");
3  }
4  public void teste2() throws FileNotFoundException {
5      teste1();
6  }
7  public void teste3() {
8      try {
9          teste2();
10     } catch(FileNotFoundException e) {
11         // ...
12     }
13 }
```

É possível tratar mais de uma exceção simultaneamente.

- Com o *try-catch*:

```
1 try {
2     objeto.metodoQuePodeLancarIOeSQLException();
3 } catch (IOException e) {
4     // ...
5 } catch (SQLException e) {
6     // ...
7 }
```

- Com a cláusula `throws`:

```
1 public void abre(String arquivo) throws IOException,
      SQLException {
2     // ..
3 }
```

- Ou mesmo uma combinação de ambos: uma exceção tratada no próprio método (*try-catch*) em combinação com a cláusula `throws`:

```
1 public void abre(String arquivo) throws IOException {
2     try {
3         objeto.metodoQuePodeLancarIOeSQLException();
4     } catch (SQLException e) {
5         // ..
6     }
7 }
```

Até o momento abordou-se o tratamento de exceções que ocorrem em métodos ou mecanismos do Java, como acesso a elementos de *arrays* e métodos relacionados a arquivos (entrada/saída).

Como **lançar** (*to throw*) as exceções em Java?

Por exemplo:

```
1 boolean saca(double valor) {
2     if (this.saldo < valor) {
3         return false;
4     } else {
5         this.saldo-=valor;
6         return true;
7     }
8 }
```

Como mostrado previamente, método acima não foi tratado por quem o chamou:

```
1 Conta minhaConta = new Conta();
2 minhaConta.deposita(100);
3 // ...
4 double valor = 5000; // valor acima do saldo (100)
      + limite
5 minhaConta.saca(valor); // saca: booleano. Vai
     retornar false, mas ninguém verifica!
6 caixaEletronico.emite(valor);
```

Podemos lançar uma exceção usando a palavra reservada `throw`. Veja como ficaria o método `saca()` visto anteriormente com o lançamento da exceção:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new RuntimeException();
4     } else {
5         this.saldo-=valor;
6     }
7 }
```

Lançamos uma exceção `RuntimeException()`, que *pode* ser tratada por quem chamou o método `saca()` (*unchecked exception*).

Desvantagem:

`RuntimeException()` – muito genérica: como saber onde exatamente ocorreu problema?

Algo mais específico:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new IllegalArgumentException();
4     } else {
5         this.saldo-=valor;
6     }
7 }
```

`IllegalArgumentException()` (subclasse de `RuntimeException()`) informa que o parâmetro passado ao método foi ruim (exemplo: valor negativo, valor acima do saldo, etc.)

No método que chamou saca(), pode-se, por exemplo, usar *try-catch* para tentar "laçar" a exceção lançada pelo mesmo:

```
1 Conta minhaConta = new Conta();
2 minhaConta.deposita(100);
3 try {
4     minhaConta.saca(100);
5 } catch (IllegalArgumentException e) {
6     System.out.println("Saldo insuficiente ou valor
          inválido.");
7 }
```

Outra forma: passar no *construtor* da exceção qual o problema – usando String:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new IllegalArgumentException("Saldo insuficiente
              ou valor inválido");
4     } else {
5         this.saldo-=valor;
6     }
7 }
```

Neste caso, o que ocorreu pode ser recuperado com o método `getMessage()` da classe `Throwable`.

```
1 try {
2     cc.saca(100);
3 } catch (IllegalArgumentException e) {
4     System.out.println(e.getMessage());
5 }
```

Observação

Todo método que lança uma exceção **verificada** *deve* conter a cláusula `throws` em sua assinatura.

```
1 public void setData(int dia, int mes, int ano) throws
      IOException {
2     if (dia<1 || dia>31) throw new IOException("Dia inválido"
          );
3     if (mes<1 || mes>12) throw new IOException("Mês inválido"
          );
4     this.dia = dia;
5     this.mes = mes;
6     this.ano = ano;
7 }
```

Criando um tipo de exceção I

É possível também criar uma **nova classe** de exceção.
Para isso, basta "estender" alguma subclasse de `Throwable`.

Voltando ao exemplo anterior:

```
1 public class SaldoInsuficienteException extends
      RuntimeException {
2     SaldoInsuficienteException(String message) {
3         super(message);
4     }
5 }
```

Ao invés de se usar `IllegalArgumentException`, pode-se lançar a exceção criada, contendo uma mensagem que dirá `Saldo insuficiente`, por exemplo:

```
1 void saca(double valor) {
2     if (this.saldo < valor) {
3         throw new SaldoInsuficienteException("Saldo
              Insuficiente," + " tente um valor menor");
4     } else {
5         this.saldo-=valor;
6     }
7 }
```

Testando:

```
1  public static void main(String[] args) {
2      Conta cc = new ContaCorrente();
3      cc.deposita(10);
4
5      try {
6          cc.saca(100);
7      } catch (SaldoInsuficienteException e) {
8          System.out.println(e.getMessage());
9      }
10 }
```

Criando um tipo de exceção V

Para transformar a exceção em *checked*, **forçando** o método que chamou `saca()` a tratar a exceção,
basta criar a exceção como subclasse de `Exception`, ao invés de `RuntimeException`:

```
1 public class SaldoInsuficienteException extends
      Exception {
2     SaldoInsuficienteException(String message) {
3         super(message);
4     }
5 }
```

`finally` I

Um bloco *try-catch* pode apresentar uma `terceira cláusula`, indicando o que deve ser feito após um `try` ou `catch` qualquer.
A ideia é, por exemplo, liberar um recurso no `finally`, como fechar um arquivo ou encerrar uma conexão com um banco de dados, independente de algo ter falhado no código: bloco **sempre** executado, independente de exceção ter ocorrido.

finally II

Exemplo:

```
1  try {
2      // bloco try
3  } catch (IOException ex) {
4      // bloco catch 1
5  } catch (SQLException sqlex) {
6      // bloco catch 2
7  } finally {
8      // bloco que será sempre executado, independente
9      // se houve ou não exception
10 }
```

Exercícios I

1. Construa um programa que leia 2 valores para que se possa fazer a divisão. No caso, crie uma exceção para tratar o problema de divisão por zero.
2. Construa um programa que crie uma classe para tratar a exceção relacionada a um caractere minúsculo em uma `String`. Faça um programa que em que dada uma `String`, possa avaliar se há um caractere minúsculo.

1. Apostila de Java e POO Caelum: disponível em https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf – acesso em: MAI/2017.
2. Documentação Java Oracle: https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

Os slides de parte desta seção foram cedidos por Marcelo Z. do Nascimento, FACOM/UFU
LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU

Há várias fontes (entrada) de onde se deseja ler, ou destinos (saída) para onde se deseja gravar ou enviar dados:

- **Arquivos**;
- Memória;
- Teclado, tela, impressora, mouse, etc.

Há várias formas diferentes de ler/escrever dados:

- Sequencialmente/aleatoriamente
- Como bytes, como caracteres
- Linha por linha, palavra por palavra, etc

Como oferecer tais serviços em Java??

Stream (fluxo) I

A linguagem Java *não* trata dispositivos de entrada e saída (E/S) de forma específica, ou seja, com classes específicas para cada dispositivo;
Ao invés disso, Java utiliza um mecanismo genérico que permite tratar E/S de forma uniforme: ***Streams* de entrada e saída**.

Stream

Um ***stream*** é um canal por onde trafegam dados entre um processo computacional e uma origem – ou destino – de dados.

Stream (fluxo) II

A **ordem** do fluxo de dados, entrada ou saída, é relevante na escolha do *stream* a ser utilizado.

- ***Stream* de entrada**: para obter informações, uma aplicação abre um *stream* de uma fonte (arquivo, *socket*, memória, etc) e lê os dados desejados;
- ***Stream* de saída**: Para enviar informações, uma aplicação abre um *stream* para um destino (arquivo, *socket*, memória) e escreve os dados.

Independentemente da fonte/destino e do tipo de informações, os algoritmos para leitura e escrita são basicamente os mesmos.

Algoritmo de leitura:

```
abre um stream
enquanto há informação
   lê informação
fecha o stream
```

Algoritmo de escrita:

```
abre um stream
enquanto há informação
   escreve informação
fecha o stream
```

Pacote `java.io`

O pacote java.io contém uma coleção de classes para trabalhar com fluxo de entrada e saída de dados; Estas classes dão suporte aos algoritmos de E/S – ou I/O, de *Input*/*Output*; As classes são divididas em dois grupos, baseadas no tipo de dados sobre os quais operam:

- `InputStream` e `OutputStream`: E/S de *bytes* – suportam leitura e gravação de 8 bits;
- `Reader` e `Writer`: E/S de caracteres (`char`) – suportam leitura e gravação de caracteres **Unicode** de 16 bits.

Estas classes são `abstratas`, ou seja, contêm um ou mais métodos para os quais não há definição. Sendo assim, objetos não podem ser criados a partir de classes abstratas.

Entrada de *bytes* – `java.io.InputStream`

Classe `java.io.InputStream`: classe abstrata para lidar com fluxos de *bytes* (**dados binários**, como imagens e sons) de entrada;

Método básico: `int read()`
Usado na leitura dos dados disponíveis em um *stream*. Note que o retorno do método é um número inteiro, indicando o *byte* lido do *stream*, ou então o número de *bytes* lidos do *stream*:

Caso não haja *bytes* disponíveis para a leitura, ou tenha ocorrido algum erro em tal processo, o retorno deste método será -1.

Classe `java.io.OutputStream`: classe abstrata para lidar com fluxos de *bytes* (**dados binários**, como imagens e sons) de saída – ou seja, **dados para gravação**;

Método básico: `void write()`

Entrada de caracteres – `java.io.Reader`

Classe `java.io.Reader`: classe abstrata para lidar com fluxos de **caracteres Unicode** de entrada;

Método básico: `int read()`
Lê um caractere de 16 bits por vez, a partir de um *stream*.

Saída de caracteres – `java.io.Writer`

Classe `java.io.Writer`: classe abstrata para lidar com fluxos de **caracteres Unicode** de saída;

Método básico: `void write()`
Grava um caractere de 16 bits por vez em um *stream* de saída.

Duas abordagens comuns para implementar a persistência de dados:

- Armazenar dados em arquivos texto;
- Usar **serialização**, permitindo gravar objetos em arquivos.

Persistência de dados

Persistência de dados consiste no armazenamento **confiável** e coerente das informações em um sistema de armazenamento de dados.

- Os dados são salvos em arquivos, separados por algum caractere único, como por exemplo `','`;
- Um arquivo texto pode ser editado e visualizado facilmente por humanos;
- Simples para fazer intercâmbio de dados entre programas diferentes.

Exemplo: um arquivo chamado `teste.dat`:

```
1 José da Silva,23,7.5
2 Márcia Bastos,20,7.0
3 Carla Pereira,18,8.5
```

Compreende a **criação** do arquivo, o **armazenamento dos dados**, e o **fechamento** do arquivo:

`FileWriter`: Estabelece a conexão com o arquivo. Usado para a saída em um arquivo, baseada em caracteres:

```
FileWriter arq = new FileWriter( nomeArq );
```

`PrintWriter`: Para escrevermos Strings no arquivo, precisamos de um objeto `PrintWriter` associado ao `FileWriter`:

```
PrintWriter out = new PrintWriter( arq );
```

Podemos então usar os métodos `print()` e `println()` da classe `PrintWriter`;

Devemos implementar o código dentro de um bloco *try/catch*, pois exceções podem ser geradas (`IOException`).

`BufferredWriter`: Esta classe permite uma saída bufferizada:
Uma operação de saída **não grava imediatamente** os dados no arquivo.

Com o método `flush()`, de tempos em tempos uma quantidade de dados é enviada para o arquivo.

Consiste na recuperação das informações armazenadas em um arquivo, para serem utilizadas por determinado programa:

`FileReader`: Estabelece a conexão com o arquivo. Uma operação de entrada lê um caractere, ou seja, trabalha *com um caractere por vez*.

```
FileReader ent = new FileReader( nomeArq );
```

`BufferedReader`: Entrada bufferizada. Uma operação de entrada lê vários caracteres de uma única vez

```
BufferedReader br = new BufferedReader (ent);
```

Método utilizado para leitura: `br.readLine()`
Este método retorna `null` quando o final do arquivo for atingido.

Exemplo – UsarArquivo.java

```java
package usararquivos;
public class UsarArquivos {
    public static void main(String[] args) {
        String nome[] = new String[3];
        int idade[] = new int[3];
        double nota[] = new double[3];
        nome[0] = "José da Silva";
        nome[1] = "Márcia Bastos";
        nome[2] = "Carla Pereira";
        idade[0] = 23;
        idade[1] = 20;
        idade[2] = 18;
        nota[0] = 7.5;
        nota[1] = 7;
        nota[2] = 8.5;
        GerenciamentoArquivos gerente = new GerenciamentoArquivos();
        gerente.escrita("teste.dat", nome, idade, nota);
        gerente.leitura("teste.dat");
    }
}
```

Exemplo – GerenciamentoArquivos.java I

```
1  package usararquivos;
2  import java.io.*;
3
4  public class GerenciamentoArquivos {
5      public void escrita(String nomeArq, String[] vet1, int[] vet2,
           double[] vet3) {
6          try {
7              FileWriter arq = new FileWriter(nomeArq);
8              PrintWriter out = new PrintWriter(arq);
9              for (int i = 0; i < vet1.length; i++) {
10                 String linha = vet1[i] + ":" + vet2[i] + ":" + vet3[i];
11                 out.println(linha);
12             }
13             out.close();
14         } catch (IOException erro) {
15             System.out.println("Erro na escrita dos dados");
16         }
17     }  //fim do método escrita()
18
19     public void leitura(String nomeArq) {
20         try {
```

Exemplo – GerenciamentoArquivos.java II

```
21            FileReader ent = new FileReader(nomeArq);
22            BufferedReader br = new BufferedReader(ent);
23            String linha;
24            String[] campos = null;
25            while ((linha = br.readLine()) != null) {
26               campos = linha.split(":");
27               String nome = campos[0];
28               int idade = Integer.parseInt((campos[1]));
29               double nota = Double.parseDouble(              campos[2].
                     replace(",", ".") );
30               System.out.println("Nome=" + nome + " Idade=" + idade +
                      " Nota=" + nota);
31            }
32            br.close();
33         } catch (IOException erro) {
34            System.out.println("Erro na leitura dos dados");
35         }
36      } // Fim do método leitura()
37   }  // Fim da classe
```

Exemplo – GerenciamentoArquivos.java III

Serialização: Processo de transformar o **estado** de um objeto em uma sequência de *bytes* que representem o valor de seus atributos:

- Obs: métodos e construtores não fazem parte da serialização;
- Após a serialização, é possível gravar o objeto serializado (sequência de *bytes*) em um arquivo, enviá-lo através da rede, etc.

Deserialização: é o processo inverso, de reconstruir um objeto a partir de uma sequência de *bytes* para o mesmo estado que o objeto estava antes de ser serializado:

- Quando os objetos forem recuperados, é preciso recriar as instâncias e reconectá-las da maneira correta.

Como permitir a serialização/deserialização em Java?

Fazendo os objetos implementarem a interface **Serializable** (do pacote `java.io`).

Serializable não tem métodos: serve apenas para indicar que os atributos destes objetos podem ser serializados e deserializados.

Passos para gravar/escrever um objeto serializado em um arquivo:

- Criar um objeto `FileOutputStream`:
  ```
  FileOutputStream arq = new FileOutputStream(nomeArq);
  ```
- Criar um objeto `ObjectOutputStream`:
  ```
  ObjectOutputStream os = new ObjectOutputStream( arq );
  ```
- Gravar o objeto:
  ```
  os.writeObject ( objeto );
  ```
- Fechar o objeto `ObjectOutputStream`:
  ```
  os.close();
  ```

Não esquecer

Para que uma classe seja serializada, ela deve implementar a interface `Serializable`

Restauração do estado de um objeto:

- Criar um objeto `FileInputStream`:
  ```
  FileInputStream arq = new FileInputStream(nomeArq);
  ```
- Criar um objeto `ObjectInputStream`:
  ```
  ObjectInputStream is = new ObjectInputStream(arq);
  ```
- Ler o objeto:
  ```
  Medicamento m=(Medicamento)is.readObject();
  ```
- Trabalhar com o objeto:
  ```
  System.out.print( "Nome: "+ m.getNome());
  ```
- Fechar o objeto `ObjectOutputStream`:
  ```
  is.close();
  ```

Exemplo – Medicamento.java I

```java
package serialfarmacia;

import java.io.Serializable;

public class Medicamento implements Serializable {

    String nome;
    double preco;

    public Medicamento() {
    }

    public Medicamento(String novoNome, double novoPreco) {
        this.nome = novoNome;
        this.preco = novoPreco;
    }

    public void setNome(String novoNome) {
        this.nome = novoNome;
    }
```

```
21
22
23      public void setPreco(double novoPreco) {
24          this.preco = novoPreco;
25      }
26
27      public String getNome() {
28          return this.nome;
29      }
30
31      public double getPreco() {
32          return this.preco;
33      }
34
35      public void escreverMedicamento() {
36          System.out.println("Nome" + this.nome);
37          System.out.println("Preco" + this.preco);
38      }
39  }
```

Exemplo – Medicamento.java III

Exemplo – TestaFarmaciaSerializacao.java I

```
1  package serialfarmacia;
2
3  public class TestaFarmaciaSerializacao {
4
5      public static void main(String[] args) {
6          Farmacia ufu = new Farmacia();
7          /* cadastro de medicamentos */
8          Medicamento m = new Medicamento("a", 5.6);
9          ufu.cadastraMedicamento(m);
10         m = new Medicamento("b", 15.6);
11         ufu.cadastraMedicamento(m);
12         m = new Medicamento("c", 25.6);
13         ufu.cadastraMedicamento(m);
14         m = new Medicamento("d", 35.6);
15         ufu.cadastraMedicamento(m);
16         m = new Medicamento("e", 3.6);
17         ufu.cadastraMedicamento(m);
18
19         //Serializa os objetos
```

```
20          ufu.escreverMedicamentos("medicamentos.dat");
21
22          //Deserializa os objetos
23          ufu.lerMedicamentos("medicamentos.dat");
24      }
25
26  }
```

Exemplo – Farmacia.java I

```
1  package serialfarmacia;
2
3  import java.io.*;
4
5  public class Farmacia {
6
7      Medicamento lista[] = new Medicamento[100];
8      int estoque = 0;
9
10     public void cadastraMedicamento(Medicamento m) {
11         lista[estoque] = m;
12         estoque++;
13     }
14
15     public void cadastrMedicamento(String nome, double preco) {
16         Medicamento m = new Medicamento(nome, preco);
17         lista[estoque] = m;
18         estoque++;
19     }
```

```
20
21
22
23      //OUTROS MÉTODOS
24      public void escreverMedicamentos() {
25          for (int i = 0; i < estoque; i++) {
26              lista[i].escreverMedicamento();
27          }
28      }
29
30      public void escreverMedicamentos(String nomeArq) {
31          try {
32              FileOutputStream arq = new FileOutputStream(nomeArq);
33              ObjectOutputStream os = new ObjectOutputStream(arq);
34              for (int i = 0; i < estoque; i++) {
35                  os.writeObject(lista[i]);
36              }
37              os.close();
38              arq.close();
39          } catch (IOException erro) {
```

```
40          System.out.println("Ocorreu um erro na escrita dos dados"
                + erro);
41      }
42  }   // Fim do método escreverMedicamentos( String )
43
44
45
46  public void lerMedicamentos(String nomeArq) {
47      try {
48          FileInputStream arq = new FileInputStream(nomeArq);
49          ObjectInputStream is = new ObjectInputStream(arq);
50          for (int i = 0; i < estoque; i++) {
51              Medicamento m = (Medicamento) is.readObject();
52              System.out.print("Nome: " + m.getNome());
53              System.out.println(" Preco: " + m.getPreco());
54          }
55          is.close(); arq.close();
56      } catch (IOException erro) {
57          System.out.println("Ocorreu um erro na escrita dos dados:
                " + erro);
```

Exemplo – Farmacia.java IV

```
58          } catch (ClassNotFoundException erro) {
59              System.out.println("Ocorreu um erro de leitura no arquivo:
                    " + erro);
60          }
61      } //Fim do método lerMedicamentos()
62  } //Fim da classe
```

Exercício I

Faça um programa que leia um arquivo de dados contendo 4 inteiros, cada inteiro correspondendo a uma quantidade de votos para um determinado time. Mostre essas informações na tela para o usuário.

1. Apostila de Java e POO Caelum: disponível em https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf – acesso em: MAI/2017.

Os slides de parte desta seção foram cedidos por Marcelo Z. do Nascimento, FACOM/UFU
LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU

A **interface gráfica com o usuário**
(abreviação: GUI – de *Graphical User Interface*)
fornece um **conjunto de componentes** facilitando a utilização de uma aplicação;

- **Botões**, **caixas de texto**, **painéis**, **barras de rolagem**, etc.

Cada componente GUI é um objeto com o qual o usuário **interage**, via mouse, teclado ou outra forma de entrada.

- **AWT** (*Abstract Window Toolkit*);
- **Swing** (mais componentes, maior flexibilidade);
- **JavaFX** (recente, coloca-se como sucessor do Swing).

Sistemas desenvolvidos com AWT são dependentes da plataforma, ou seja, em plataformas diferentes as interfaces gráficas podem ser exibidas de forma diferente, pois AWT usa as primitivas gráficas de cada plataforma;

Não fornece aparência e comportamento consistentes para diversas plataformas.

Objetivo: dotar uma aplicação Java com componentes GUI padronizados;

- Mesma aparência (ou semelhante) em qualquer Sistema Operacional.

Para isso ocorrer, a maior parte dos componentes Swing são escritos, manipulados e exibidos completamente em Java.

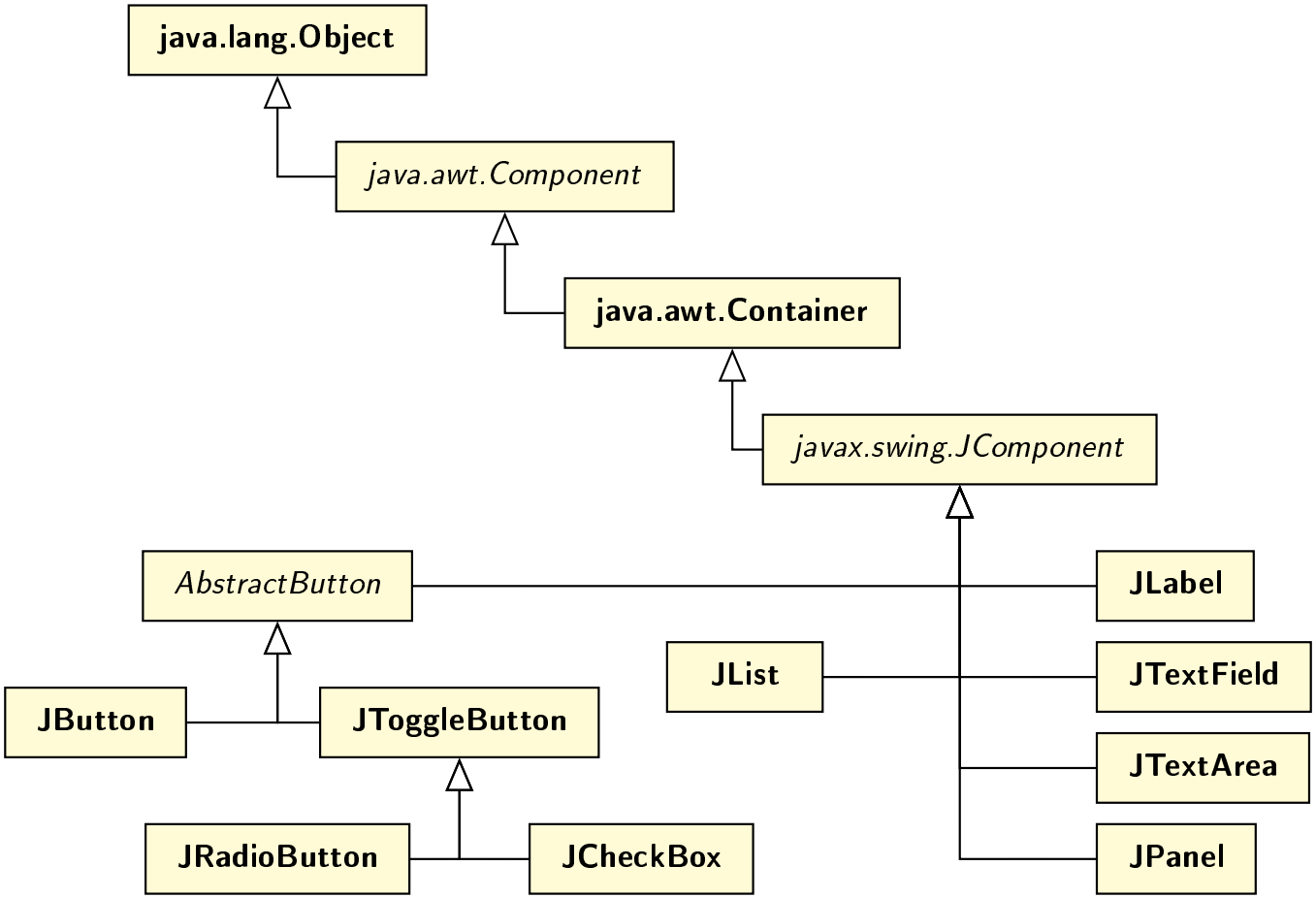
As instruções mostram como importar o pacote principal para aplicações Swing:

```
import javax.swing.*;
import javax.swing.event.*;
```

A maioria das aplicações Swing também precisam de dois pacotes AWT:

```
import java.awt.*;
import java.awt.event.*;
```

Pacote Swing III

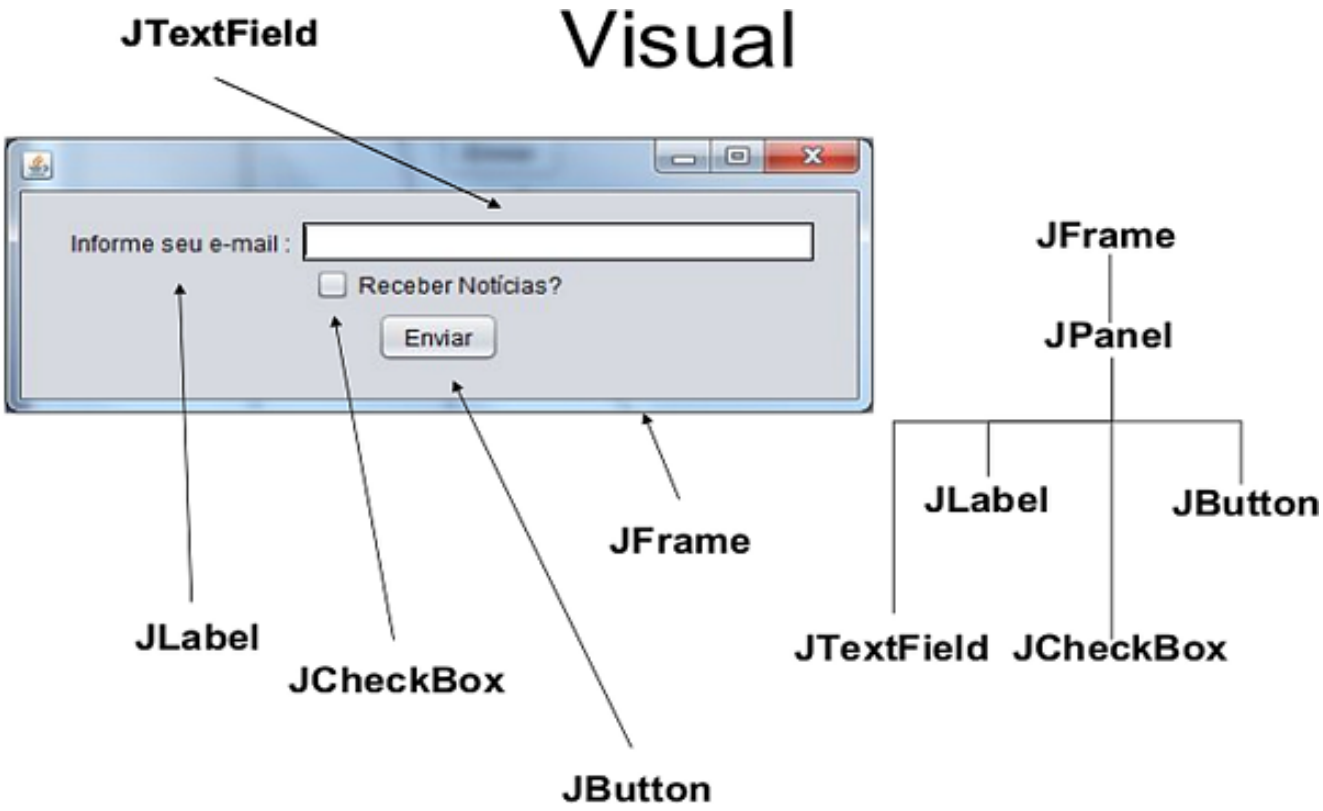


As hierarquias de herança dos pacotes `javax.swing` e `java.awt` devem ser compreendidas – especificamente a classe `Component`, a classe `Container` e a classe `JComponent`, que definem os recursos comuns à maioria dos componentes Swing:

- `Component` – define métodos que podem ser usados nas suas subclasses;
- `Container` – coleção de componentes relacionados:
  - Quando usado com JFrames insere componentes para o painel (um `Container`);
  - Método `add`.
- `JComponent` – superclasse da maioria dos componentes Swing.
  - Muitas das funcionalidades dos componentes são herdadas dessa classe.

Alguns componentes do Swing:

Componente	Descrição
`JLabel`	Área para exibir texto não-editável.
`JTextField`	Área em que o usuário insere dados pelo teclado. Também podem exibir informações.
`JButton`	Área que aciona um evento quando o mouse é pressionado.
`JCheckBox`	Componentes GUI que têm dois estados: selecionado ou não.
`JComboBox`	Lista de itens a partir da qual o usuário pode fazer uma seleção clicando em um item.
`JList`	Área em que uma lista de itens é exibida, a partir da qual o usuário pode fazer uma seleção clicando uma vez em qualquer elemento.
`JPanel`	*Container* em que os componentes podem ser adicionados.


JTextField
Visual
Informe seu e-mail :
Receber Notícias?
Enviar
JFrame
JLabel
JCheckBox
JButton
JFrame
JPanel
JLabel
JButton
JTextField
JCheckBox

Para que um componente tenha certo comportamento quando ocorre um **evento** os passos abaixo devem ser seguidos:

1. Criação de uma classe que estenda `JFrame` (define uma janela do sistema);
2. Criação do componente visual;
3. Adição desse componente em um contêiner;
4. Criação de uma classe interna (tratador de eventos) que implemente determinado *listener*;
5. Registro do tratador de eventos ao *listener* através dos métodos disponíveis nas instâncias dos componentes GUI.

Observação: Para os componentes que não possuem eventos associados (ex.: `JLabel`), somente os 3 primeiros passos são aplicáveis.

Exemplo com classe JFrame:

```
1  import javax.swing.*;
2  public class Exemplo1 extends JFrame  {
3      public Exemplo1() {
4          // Define o título da janela
5          super("Primeira janela");
6
7          this.setSize(320, 240); // os métodos setSize() e
8          this.setVisible(true);  // setVisible são obrigatórios
               .
9      }
10
11     public static void main(String[] args) {
12         Exemplo1 janela = new Exemplo1();
13     }
14 }
```

Swing – exemplos básicos II

Exemplo com classe JFrame:

```
1  import javax.swing.*;
2  import java.awt.event.*;
3  public class Exemplo2 extends JFrame {
4
5      public Exemplo2() {
6          // Define o título da janela
7          super("Primeira janela");
8
9          this.setSize(320, 240);
10         this.setVisible(true);
11     }
12
13     public static void main(String[] args) {
14         Exemplo2 janela = new Exemplo2();
15
```

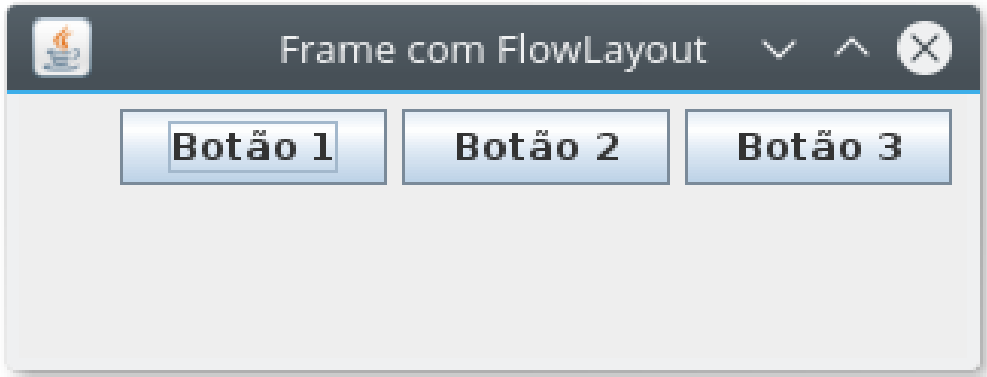
```
16          // Quando janela é fechada, apenas se torna invisível.
17          // Com comandos a seguir, será chamado o método exit()
                ,
18          // que encerra a aplicação e libera a JVM.
19          janela.addWindowListener(
20              new WindowAdapter() { // classe do pacote awt.event
21                  public void windowClosing(WindowEvent e) {
22                      System.exit(0);
23                  }
24              }
25          );
26      }
27  }
```

Os **gerenciadores de *layout*** organizam os componentes GUI em um contêiner para fins de apresentação.
Os principais gerenciadores de *layout* são:

- `FlowLayout` – componentes dispostos em linha, da esquerda para a direita, na ordem em que foram adicionados.
- `BorderLayout` – componentes dispostos em 5 regiões: `NORTH`, `SOUTH`, `EAST`, `WEST`, `CENTER` (cada região: máximo 1).
- `GridLayout` – Área dividida em retângulos, conforme número de linhas/colunas especificados.
- `SpringLayout` – combina características dos demais; baseia-se nas relações ou restrições entre as bordas dos componentes.

```
1  import java.awt.*;
2  import javax.swing.*;
3  import java.awt.event.*;
4
5  public class Exemplo3 extends JFrame {
6      public Exemplo3() {
7          super("Frame com FlowLayout");
8          JButton b1 = new javax.swing.JButton("Botão 1");
9          JButton b2 = new JButton("Botão 2");
10         JButton b3 = new JButton("Botão 3");
11         this.setSize(320, 120);
12         Container c = this.getContentPane();
13         c.add(b1);
14         c.add(b2);
15         c.add(b3);
16         c.setLayout(new FlowLayout(FlowLayout.RIGHT));
```

```
17          this.setVisible(true);
18      }
19
20      // método main() aqui, como no exemplo anterior
21      // ...
22  }
```


Frame com FlowLayout
Botão 1
Botão 2
Botão 3

Swing – rótulos I

- Os **rótulos** fornecem instruções de texto ou informações em um GUI;
- Os rótulos são definidos com a classe `JLabel`;
- O rótulo exibe uma **única linha de texto somente de leitura**, uma **imagem**, ou **ambos**;

Exemplo:
Desenvolver um aplicativo Java que apresente um *label* (rótulo) no rodapé de uma tela, colocando um ícone no centro da tela, associado a uma figura de sua escolha e indicando uma *frase de dica* para o usuário, que aparecerá quando o mouse repousar sobre a figura.

```
1  package labelteste;
2
3  import java.awt.*;
4  import java.awt.event.*;
5  import javax.swing.*;
6
7  public class LabelTeste extends JFrame {
8     private final JLabel label;
9     private final Icon icone = new ImageIcon( "java.jpg" );
10    // Configurando a GUI
11    public LabelTeste() {
12       super( "Testando JLabel" );
13       // Cria um container e define o modelo de layout (FlowLayout)
14       Container container = getContentPane();
15       container.setLayout( new FlowLayout() );
16       // JLabel sem argumentos no construtor
17       label = new JLabel();
```

```
18          label.setText("Label com ícone e texto com alinhamento de
                rodapé bottom" );
19          label.setIcon( icone );
20          label.setHorizontalTextPosition( SwingConstants.CENTER );
21          label.setVerticalTextPosition( SwingConstants.BOTTOM );
22          label.setToolTipText("Este é o label" );
23          container.add( label );
24          setSize( 500, 300 );
25          setVisible( true );
26       }
27
28       // Método principal da aplicação
29       public static void main( String args[] )  {
30          LabelTeste application = new LabelTeste();
31          application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
32       }
33    }  // final da classe
```


Testando JLabel
Este é o label
Java
Label com ícone e texto com alinhamento de rodapé bottom

Eventos são *mapeamentos* da **interação** do usuário com o programa.

- As GUIs são baseados em eventos, isto é, geram eventos quando o usuário interage com a interface;
- Algumas interações: *mover o mouse*, *clicar no mouse*, *clicar em um botão*, *digitar num campo de texto*, *selecionar um item de menu*, *fechar uma janela*, etc;
- Os eventos da GUI são enviados para o programa quando ocorre uma interação com o usuário;

- O mecanismo de tratamento de eventos possui três partes:
  - A origem do evento.
  - O objeto do evento.
  - O "ouvinte" (*listener*) do evento.
- A **origem** do evento é o componente GUI com o qual o usuário interage;
- O **objeto evento** encapsula as informações sobre o evento que ocorreu. As informações incluem uma referência para a origem do evento e quaisquer informações específicas que possam ser requeridas pelo *listener*;
- O ***listener*** recebe notificações de que um evento ocorreu permitindo que este realize determinada ação;

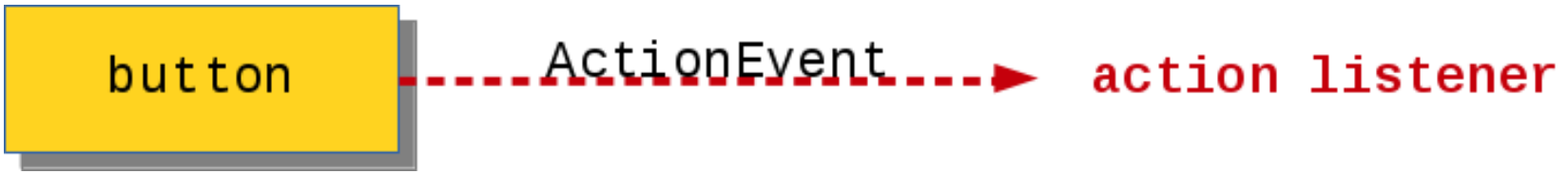
- É preciso executar as seguintes tarefas para processar um evento da GUI com o usuário em um programa:
  - registrar um *listener* para determinado componente GUI;
  - implementar um método de tratamento do evento, também chamado de **tratador de eventos** (*handler*).
- O objeto da GUI gera um `ActionEvent` (evento);
- O evento é processado por um objeto `ActionListener` (ouvinte)

- É preciso registrar um objeto `ActionListener` na lista das ações a serem executadas. Por exemplo:

```
1 buttonCadastrar.addActionListener (
2     new ActionListener() {
3         public void actionPerformed(ActionEvent e) {
4             JOptionPane.showMessageDialog(null, "Você clicou no
                  botão!");
5         }
6     }
7 );
```

Exemplo:

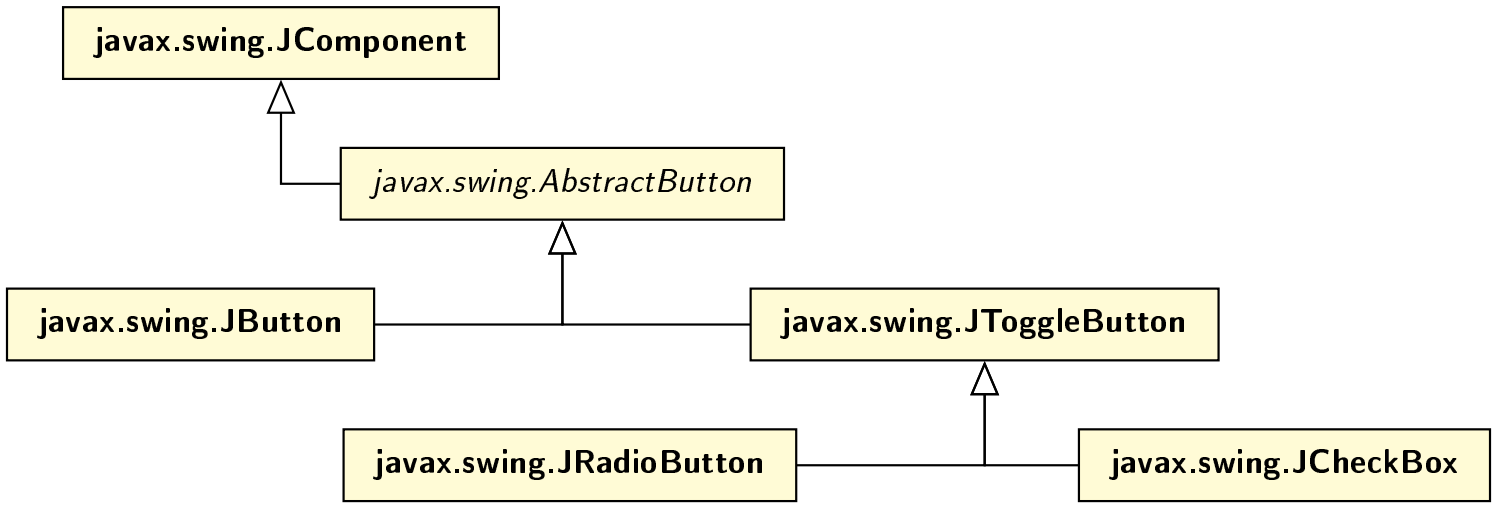
- Pressione um `JButton`;
- Método `actionPerformed` é chamado na escuta registrada para o objeto.



Como o tratador de eventos foi registrado?

O registro ocorre com os comandos:

O **botão** é um componente em que o usuário clica para disparar uma ação específica. O programa Java pode utilizar vários tipos de botões, incluindo **botões de comando**, **caixas de marcação**, **botões de alternância** e **botões de opção**.



Todos são subclasses de `AbstractButton` (pacote `javax.swing`), que define muitos dos recursos comuns aos botões do Swing.

Exemplo:

Desenvolver um aplicativo Java que apresente um botão associado a um ícone (com figura de sua escolha) e indicando uma frase de dica para o usuário (ex.: "pressione o botão"), um botão de finalização do programa e um mecanismo de tratamento do evento associado ao botão com o ícone (onde o tratamento seja apresentar uma nova janela com uma mensagem).

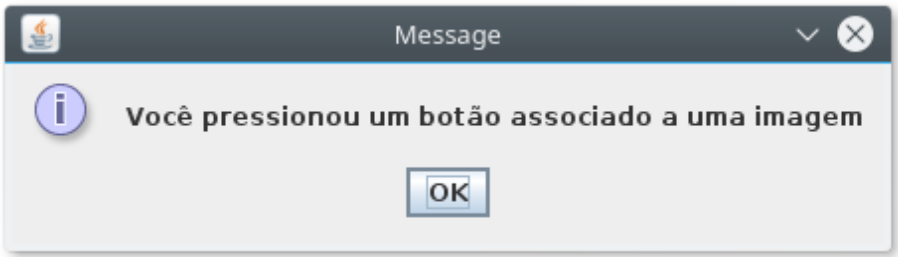
```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class ExemploButton extends JFrame {
6
7     private JButton botao1, botao2;
8     private Icon cafe = new ImageIcon("java.jpg");
9     private String strIcone = "botão associado a uma imagem";
10    private String strFinalizar = "Finalizar";
11
12    // Configura a GUI
13    public ExemploButton() {
14       super("Testando Botões");
15
16       // Cria o container e atribui o layout
17       Container container = getContentPane();
18       container.setLayout(new FlowLayout());
19
```

```
20          // Cria os botões
21          botao1 = new JButton("Botão Java", cafe);
22          botao1.setIcon(cafe);
23
24          botao1.setToolTipText("Pressione o botão");
25          botao1.setActionCommand(strIcone);
26          container.add(botao1);
27          botao2 = new JButton(strFinalizar);
28          botao2.setToolTipText("Finaliza o programa");
29          container.add(botao2);
30          // Cria o objeto gestorBotoes (instância da classe interna
                ButtonHandler)
31          // para o uso no tratamento de eventos de botão
32          GerenciadorBotoes gestorBotoes = new GerenciadorBotoes();
33          botao1.addActionListener(gestorBotoes);
34          botao2.addActionListener(gestorBotoes);
35
36          setSize(545, 280);
37          setVisible(true);
38      }
```

```
39
40      // Classe interna para tratamento de evento de botão
41      private class GerenciadorBotoes implements ActionListener {
42          // Método de manipulação do evento
43
44          public void actionPerformed(ActionEvent event) {
45              //Testa se o botão com a imagem foi pressionado
46              if (event.getActionCommand().equalsIgnoreCase(strIcone)) {
47                  JOptionPane.showMessageDialog(null,
48                  "Você pressionou um " + event.getActionCommand());
49              } //Testa se o botão "Finalizar" foi pressionado
50              else if (event.getActionCommand().equalsIgnoreCase(
                        strFinalizar)) {
51                  System.exit(0);
52              }
53          }
54      }  // fim da classe interna GerenciadorBotoes
55
56      // Método principal
57      public static void main(String args[]) {
```

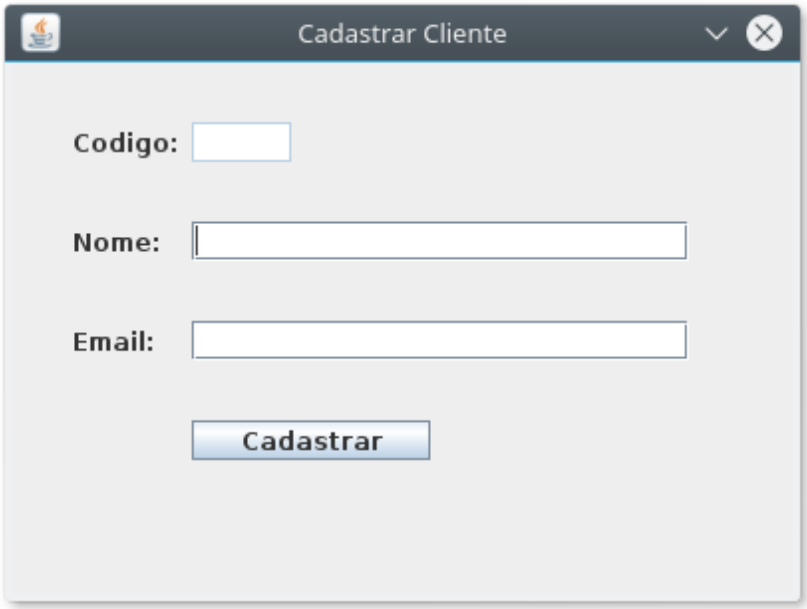
```
58          ExemploButton application = new ExemploButton();
59          application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
60      }
61
62  }   // fim da classe ExemploButton
```


Testando Botões
Java
Botão Java
Finalizar


Message
Você pressionou um botão associado a uma imagem
OK

Exemplo:

Desenvolver um aplicativo Java semelhante à figura abaixo:



```
1  import javax.swing.*;
2
3  public class FCliente extends JFrame {
4      private JLabel labelCodigo;
5      private JLabel labelNome;
6      private JLabel labelEmail;
7      private JTextField fieldCodigo;
8      private JTextField fieldNome;
9      private JTextField fieldEmail;
10     private JButton buttonCadastrar;
11
12     public FCliente() {
13         initComponents();
14     }
15
16     private void initComponents() {
```

```
17        labelCodigo = new JLabel();
18        labelNome = new JLabel();
19        labelEmail = new JLabel();
20        fieldCodigo = new JTextField();
21
22        fieldNome = new JTextField();
23        fieldEmail = new JTextField();
24        buttonCadastrar = new JButton();
25        this.setTitle("Cadastrar Cliente");
26        this.setSize(400, 300);
27        this.setResizable(false);
28        this.setDefaultCloseOperation(WindowConstants.
              EXIT_ON_CLOSE);
29        this.getContentPane().setLayout(null);
30        labelCodigo.setText("Codigo:");
31        labelCodigo.setBounds(30, 30, 70, 20);
32        this.add(labelCodigo);
```

```
33          labelNome.setText("Nome:");
34          labelNome.setBounds(30, 80, 70, 20);
35          this.add(labelNome);
36          labelEmail.setText("Email:");
37          labelEmail.setBounds(30, 130, 70, 20);
38          this.add(labelEmail);
39          fieldCodigo.setBounds(90, 30, 50, 20);
40          fieldCodigo.setEnabled(false);
41          this.add(fieldCodigo);
42          fieldNome.setBounds(90, 80, 250, 20);
43          this.add(fieldNome);
44          fieldEmail.setBounds(90, 130, 250, 20);
45          this.add(fieldEmail);
46          buttonCadastrar.setText("Cadastrar");
47          buttonCadastrar.setBounds(90, 180, 120, 20);
48          this.add(buttonCadastrar);
49          this.setVisible(true);
```

```
50      }
51
52      public static void main(String[] args) {
53          //Schedule a job for the event-dispatching thread:
54          //creating and showing this application's GUI.
55          javax.swing.SwingUtilities.invokeLater(new Runnable()
                {
56              public void run() {
57                  new FCliente();
58              }
59          });
60      }
61  } // fim da classe
```

Exemplo:

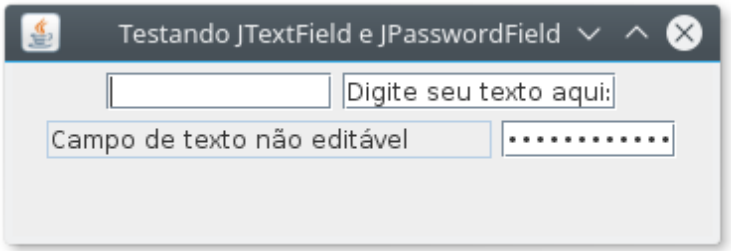
Desenvolver um aplicativo Java que apresente quatro campos de edição, sendo um para o usuário colocar uma *frase*, outro para apresentar uma *frase editável*, outro para apresentar um *texto não-editável* e um último para *registrar senhas*. Trate os eventos associados ao acionamento da tecla Enter em cada um desses campos de edição.

```java
// Exemplo de JTextField
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ExemploJTextField extends JFrame {
   private JTextField campoTexto1, campoTexto2, campoTexto3;
   private JPasswordField campoSenha;

   // configuração da GUI
   public ExemploJTextField() {
      super("Testando JTextField e JPasswordField");
      Container container = getContentPane();
      container.setLayout(new FlowLayout());

      // constrói o 1o campo de texto com dimensões default
      campoTexto1 = new JTextField(10);
      container.add(campoTexto1);

      // constrói o 2o campo de texto com texto default
```

```
21        campoTexto2 = new JTextField("Digite seu texto aqui:");
22        container.add(campoTexto2);
23
24        // constrói o 3o campo de texto com texto default e
25        // 20 elementos visíveis, sem tratador de eventos
26        campoTexto3 = new JTextField("Campo de texto não editável",
              20);
27        campoTexto3.setEditable(false);
28        container.add(campoTexto3);
29
30        // constrói o 4o campo de texto com texto default
31        campoSenha = new JPasswordField("Texto oculto");
32        container.add(campoSenha);
33        // registra os tratadores de evento
34        GerenciadorTextField gerenteTexto = new GerenciadorTextField
              ();
35        campoTexto1.addActionListener(gerenteTexto);
36        campoTexto2.addActionListener(gerenteTexto);
37        campoTexto3.addActionListener(gerenteTexto);
38        campoSenha.addActionListener(gerenteTexto);
```

```
39          setSize(360, 120);
40          setVisible(true);
41      }
42
43      public static void main(String args[]) {
44          ExemploJTextField programaTexto = new ExemploJTextField();
45          programaTexto.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
46      }
47
48      //classe interna privativa para tratamento de eventos
49      private class GerenciadorTextField implements ActionListener {
50          //processa eventos de campos de texto
51          public void actionPerformed(ActionEvent evento) {
52              String texto = "";
53              // evento:Usuário pressiona ENTER no objeto de JTextField
                    campoTexto1
54              if (evento.getSource() == campoTexto1) {
55                  texto = "campoTexto1: " + evento.getActionCommand();
56              } // evento:Usuário pressiona ENTER no objeto de
                    JTextField campoTexto2
```

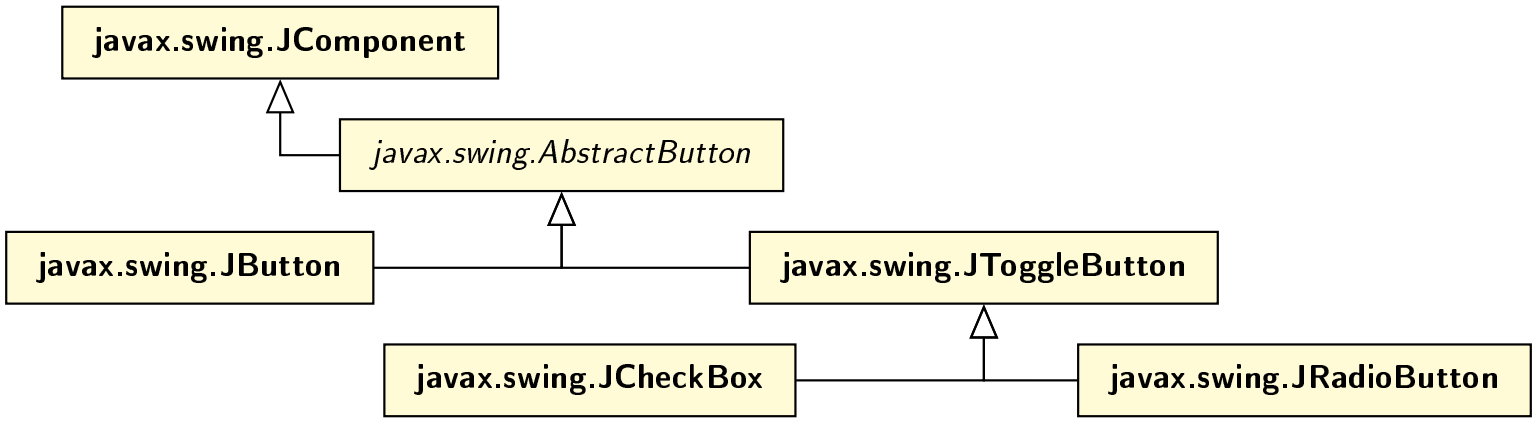
```
57            else if (evento.getSource() == campoTexto2) {
58                texto = "campoTexto2: " + evento.getActionCommand();
59            } // evento:Usuário pressiona ENTER no objeto de
                  JTextField campoTexto3
60            else if (evento.getSource() == campoTexto3) {
61                texto = "campoTexto3: " + evento.getActionCommand();
62            } // evento:Usuário pressiona ENTER no objeto de
                  JPasswordField campoSenha
63            else if (evento.getSource() == campoSenha) {
64                JPasswordField senha = (JPasswordField) evento.
                      getSource();
65                texto = "campoSenha: " + new String(campoSenha.
                      getPassword());
66            }
67            JOptionPane.showMessageDialog(null, texto);
68        }
69    } // fim da classe interna privativa GerenciadorTextField
70 } // fim da classe ExemploJTextField
```


Testando JTextField e JPasswordField
Digite seu texto aqui:
Campo de texto não editável

Para que o usuário interaja com um aplicativo Java, existem diversos tipos de botões para cada situação de interface.

Os componentes GUI Swing possuem três tipos de botões de estado (que assumem valores ativados/desativados ou verdadeiro/falso):

- `JToggleButton` – para barras de ferramentas;
- `JCheckBox` – para interfaces de múltipla escolha;
- `JRadioButton` – escolha única entre múltiplas alternativas.



Exemplo:

Aplicativo Java que permita que o usuário digite uma frase e veja sua sentença aparecer em **negrito**, *itálico* ou em ***ambos***, dependendo de sua escolha.

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class ExemploCheckBoxRadio extends JFrame {
6
7     private JCheckBox checkB, checkI;
8     private JRadioButton rbotao1, rbotao2, rbotao3;
9     private ButtonGroup grupoRadio;
10    private JPanel painel1, painel2;
11
12    // Configura a GUI
13    public ExemploCheckBoxRadio() {
14       super("Testando CheckBox e RadioButton");
15
16       // Cria o container e atribui o layout
17       Container container = getContentPane();
18       container.setLayout(new FlowLayout());
```

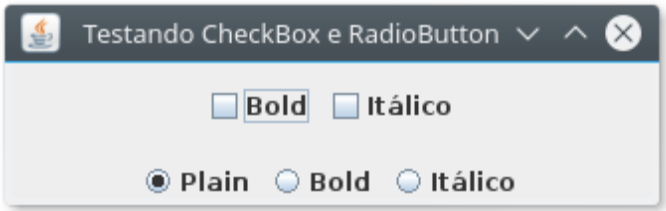
```
19        // Cria os painéis
20        painel1 = new JPanel();
21        painel2 = new JPanel();
22
23        // Cria os objetos CheckBox, adiciona para o painel e
24        // adiciona o painel para o container
25        checkB = new JCheckBox("Bold");
26        painel1.add(checkB);
27        checkI = new JCheckBox("Itálico");
28        painel1.add(checkI);
29        container.add(painel1);
30
31        // Cria os objetos RadioButton, adiciona para o painel e
32        //adiciona o painel para o container
33        rbotao1 = new JRadioButton("Plain", true);
34        painel2.add(rbotao1);
35        rbotao2 = new JRadioButton("Bold", false);
36        painel2.add(rbotao2);
37        rbotao3 = new JRadioButton("Itálico", false);
38        painel2.add(rbotao3);
```

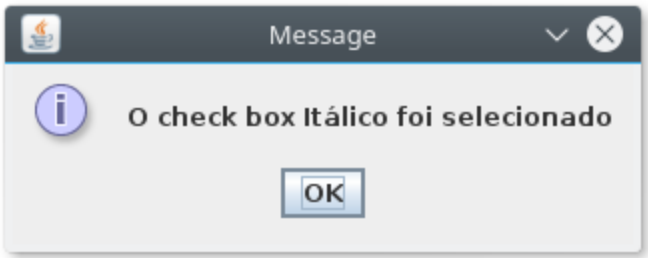
```
39          container.add(painel2);
40
41          //Cria o relacionamento lógico entre os objetos JRadioButton
42          grupoRadio = new ButtonGroup();
43          grupoRadio.add(rbotao1);
44          grupoRadio.add(rbotao2);
45
46          grupoRadio.add(rbotao3);
47
48          //Registra os tratadores de evento
49          Gerenciador gerente = new Gerenciador();
50          checkB.addItemListener(gerente);
51          checkI.addItemListener(gerente);
52          rbotao1.addItemListener(gerente);
53          rbotao2.addItemListener(gerente);
54          rbotao3.addItemListener(gerente);
55
56          setSize(300, 100);
57          setVisible(true);
58      }
```

```
59
60      // Classe interna para tratamento de evento
61      private class Gerenciador implements ItemListener {
62          // Método de manipulação do evento
63
64          public void itemStateChanged(ItemEvent event) {
65              //Testa qual objeto foi pressionado
66              if (event.getSource() == checkB) {
67
68
69                  JOptionPane.showMessageDialog(null, "O check box Bold
                        foi selecionado");
70              } else if (event.getSource() == checkI) {
71                  JOptionPane.showMessageDialog(null, "O check box Itá
                        lico foi selecionado");
72              } else if ((event.getSource() == rbotao1)
73                  && (event.getStateChange() == ItemEvent.SELECTED)) {
74                      JOptionPane.showMessageDialog(null, "O radio button
                            Plain foi selecionado");
75              } else if ((event.getSource() == rbotao2)
```

```
76              && (event.getStateChange() == ItemEvent.SELECTED)) {
77                  JOptionPane.showMessageDialog(null, "O radio button
                        bold foi selecionado");
78          } else if ((event.getSource() == rbotao3)
79              && (event.getStateChange() == ItemEvent.SELECTED)) {
80                  JOptionPane.showMessageDialog(null, "O radio button
                        Itálico foi selecionado");
81          }
82      }
83  }  // fim da classe interna Gerenciador
84
85
86  // Método principal
87  public static void main(String args[]) {
88      ExemploCheckBoxRadio application = new ExemploCheckBoxRadio()
            ;
89      application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
90  }
```

```
91 }   // fim da classe ExemploCheckBoxRadio
```


Testando CheckBox e RadioButton
Bold
Itálico
Plain
Bold
Itálico


Message
O check box Itálico foi selecionado
OK

Os **menus** também são parte integrante das GUIs, permitindo que a interface fique mais organizada:

- As classes utilizadas para definir menus são `JMenuBar`, `JMenuItem`, `JCheckBoxMenuItem` e `JRadioButtonMenuItem`.
- A classe `JMenuBar` contém os métodos necessários para gerenciar uma barra de menus;
- A classe `JMenu`, por sua vez, contém os métodos necessários para gerenciar menus;
- Os menus contêm itens e são adicionados à barra de menus;
- A classe `JItemMenu` contém os métodos necessários para gerenciar os itens dos menus;
- O item de menu é um componente GUI pertencente ao componente menu que quando selecionado realiza determinada ação;

- A classe `JCheckBoxMenuItem` contém os métodos necessários para gerenciar itens de menu que podem ser ativados ou desativados;
- A classe `JRadioButtonMenuItem` contém os métodos necessários para gerenciar itens de menu que também podem ser ativados ou desativados.

Exemplo:

Desenvolver um aplicativo Java que apresente três menus: **Cadastro**, **Relatórios** e **Ajuda**, na barra superior da janela. O primeiro menu deve possibilitar o cadastro de Paciente e Médicos, e permitir que o sistema seja finalizado. O terceiro menu deve ter um item que possibilite a visualização de um tela com informações do sistema (**Sobre**).

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class ExemploMenu extends JFrame {
6
7     private JMenuBar barraMenu;
8     private JMenu mCad, mRel, mAjuda;
9     private JMenuItem iPac, iMed, iFim, iSobre;
10    private String sistema = "Sistema de Gerenciamento de Clínicas";
11    private String versao = "Versao 1.0";
12    private String build = "(build 20030626)";
13
14    // Configura a GUI
15    public ExemploMenu() {
16       //Atribui o título para a janela
17       setTitle(sistema);
18
19       //Cria a barra de menus
```

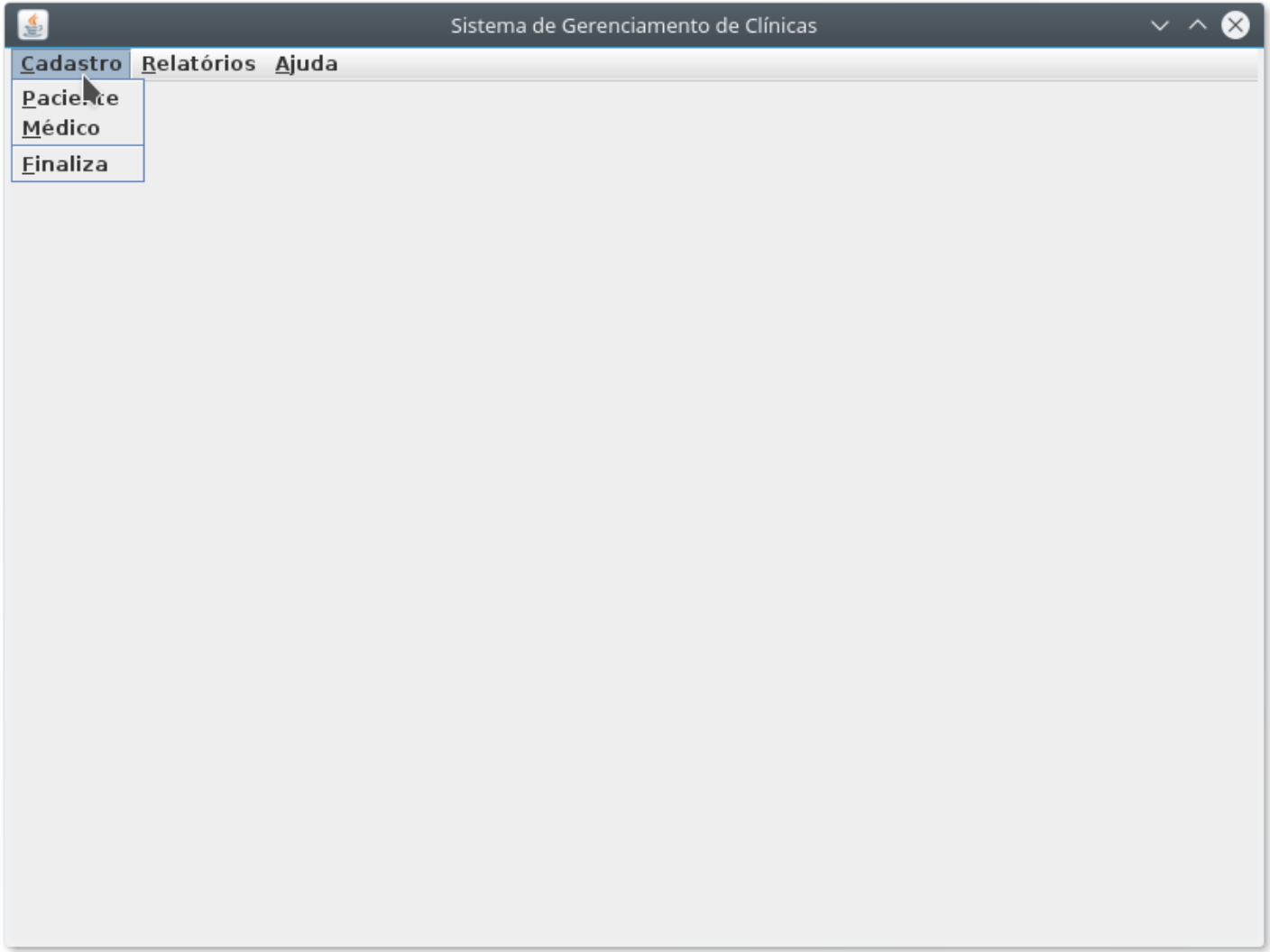
```
20        barraMenu = new JMenuBar();
21
22
23        //Cria os menus e adiciona para a barra
24        mCad = new JMenu("Cadastro");
25        mCad.setMnemonic('C');
26        mRel = new JMenu("Relatórios");
27        mRel.setMnemonic('R');
28        mAjuda = new JMenu("Ajuda");
29        mAjuda.setMnemonic('A');
30        barraMenu.add(mCad);
31        barraMenu.add(mRel);
32        barraMenu.add(mAjuda);
33        //Cria os itens de menu
34        iPac = new JMenuItem("Paciente");
35        iPac.setMnemonic('P');
36        iMed = new JMenuItem("Médico");
37        iMed.setMnemonic('M');
38        iFim = new JMenuItem("Finaliza");
39        iFim.setMnemonic('F');
```

```
40        iSobre = new JMenuItem("Sobre");
41        iSobre.setMnemonic('S');
42        //Adiciona os itens para o menu de Cadastro
43        mCad.add(iPac);
44        mCad.add(iMed);
45        mCad.addSeparator();
46
47        mCad.add(iFim);
48        //Adiciona o iten sobre para o menu Ajuda
49        mAjuda.add(iSobre);
50
51        // registra os tratadores de evento
52        Gerenciador gerente = new Gerenciador();
53        iPac.addActionListener(gerente);
54        iFim.addActionListener(gerente);
55        iSobre.addActionListener(gerente);
56
57        //Anexa a barra de menu a janela
58        setJMenuBar(barraMenu);
59        setSize(800, 600);
```

```
60          setVisible(true);
61          //Configura para permitir o fechamento da aplicação
62          //quando a janela for fechada
63          setDefaultCloseOperation(EXIT_ON_CLOSE);
64      }
65
66      private class Gerenciador implements ActionListener {
67          //processa eventos de campos de texto
68
69
70          public void actionPerformed(ActionEvent evento) {
71              if (evento.getSource() == iPac) {
72                  //ExemploGridBagLayout cadastro = new
                        ExemploGridBagLayout();
73              } else if (evento.getSource() == iFim) {
74                  System.exit(0);
75              } else if (evento.getSource() == iSobre) {
76              JOptionPane.showMessageDialog(null,
77                      sistema + "\n\n" + "                " + versao + " " +
                            build + "\n\n",
```

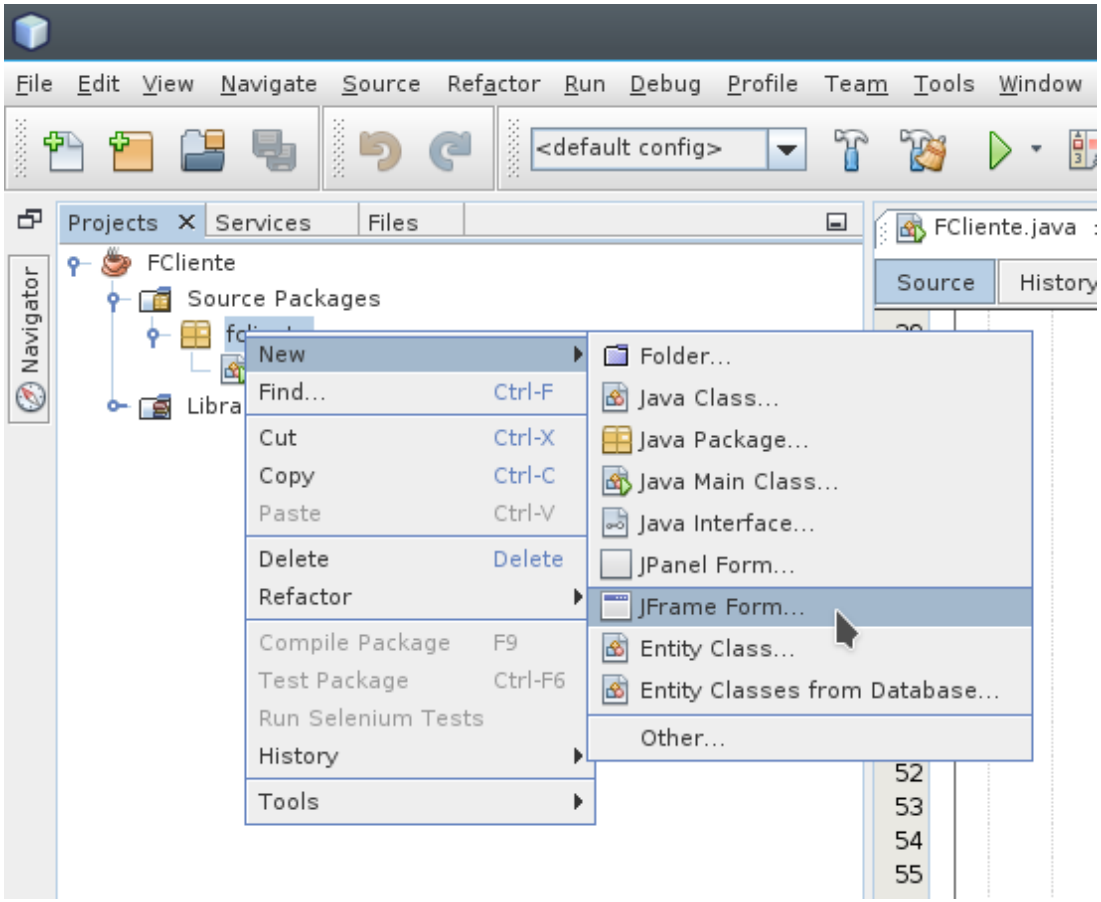
```
78                  "Sobre o sistema", JOptionPane.PLAIN_MESSAGE);
79          }
80      }
81  }
82
83      public static void main(String arg[]) {
84          ExemploMenu menuGeral = new ExemploMenu();
85      }
86
87  }   //Fim da classe ExemploMenu
```

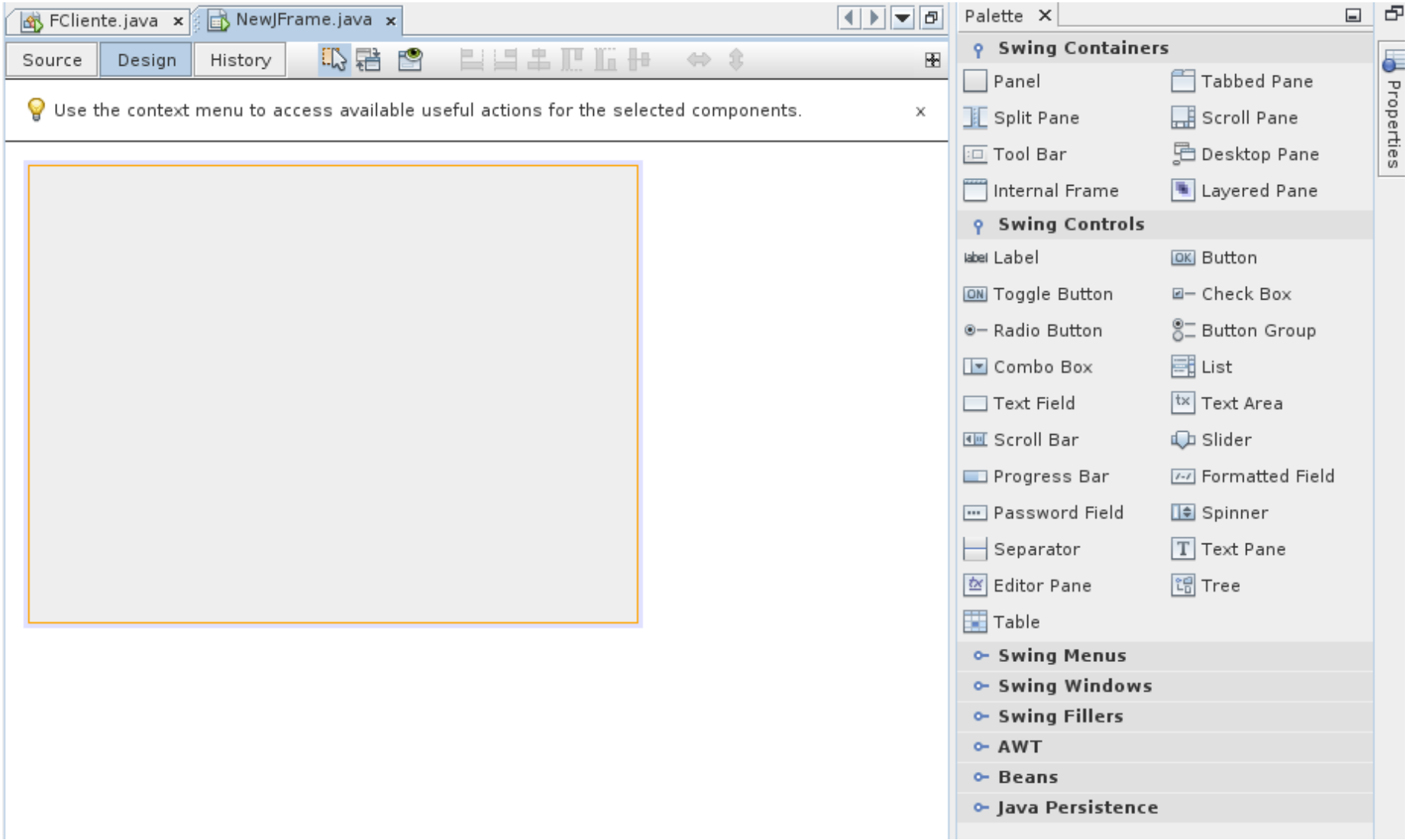
Swing – menus IX



Como visto no código anterior, o método `setMnemonic(char)` permite definir o caractere de atalho utilizado conjuntamente com a tecla Alt.

No NetBeans


File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window
<default config>
Projects Services Files
FCliente
Source Packages
Navigator
FCliente.java
Source History
New
Find... Ctrl-F
Cut Ctrl-X
Copy Ctrl-C
Paste Ctrl-V
Delete Delete
Refactor
Compile Package F9
Test Package Ctrl-F6
Run Selenium Tests
History
Tools
Folder...
Java Class...
Java Package...
Java Main Class...
Java Interface...
JPanel Form...
JFrame Form...
Entity Class...
Entity Classes from Database...
Other...
52
53
54
55


FCliente.java
NewJFrame.java
Source
Design
History
Use the context menu to access available useful actions for the selected components.
Palette
Swing Containers
Panel
Tabbed Pane
Split Pane
Scroll Pane
Tool Bar
Desktop Pane
Internal Frame
Layered Pane
Swing Controls
Label
Button
Toggle Button
Check Box
Radio Button
Button Group
Combo Box
List
Text Field
Text Area
Scroll Bar
Slider
Progress Bar
Formatted Field
Password Field
Spinner
Separator
Text Pane
Editor Pane
Tree
Table
Swing Menus
Swing Windows
Swing Fillers
AWT
Beans
Java Persistence
Properties

Os slides de parte desta seção foram cedidos por Marcelo Z. do Nascimento, FACOM/UFU
LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU
Veja também:

- https://docs.oracle.com/javase/tutorial/uiswing/start/index.html
- https://netbeans.org/kb/docs/java/quickstart-gui_pt_BR.html#project (GUI NetBeans)
- https://netbeans.org/kb/docs/java/gui-image-display_pt_BR.html (tratando imagens GUI NetBeans)
- http://slideplayer.com.br/slide/10378301 (eventos em Swing)