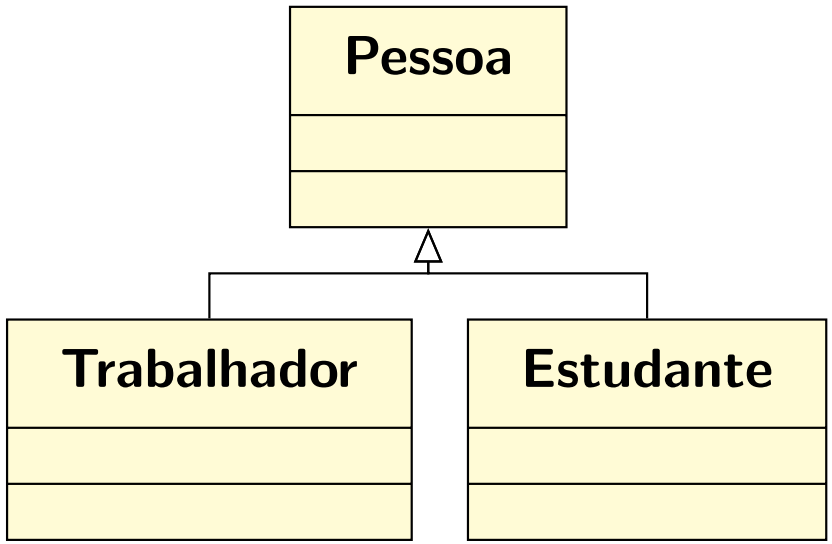
INE5603 Introdução à POO

Prof. A. G. Silva

30 de outubro de 2017

Especialização de classes (I) (Cap. 6)

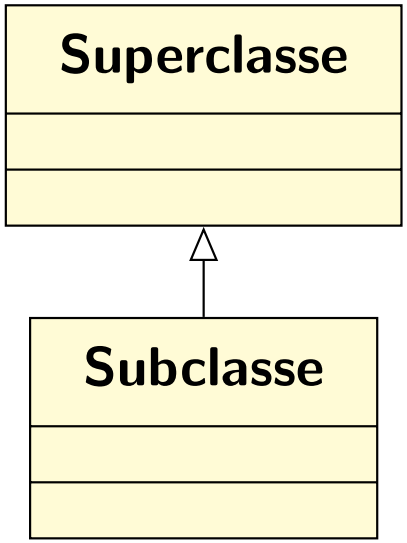
- Operações de abstração como forma de organizar as entidades
- Exemplo de especialização (relação "é um tipo de"):



- Em relação à classe *Pessoa*, as classes *Trabalhador* e *Estudante* são denominadas subclasses, ou classes derivadas, ou também classes descendentes

Especialização de classes (II)

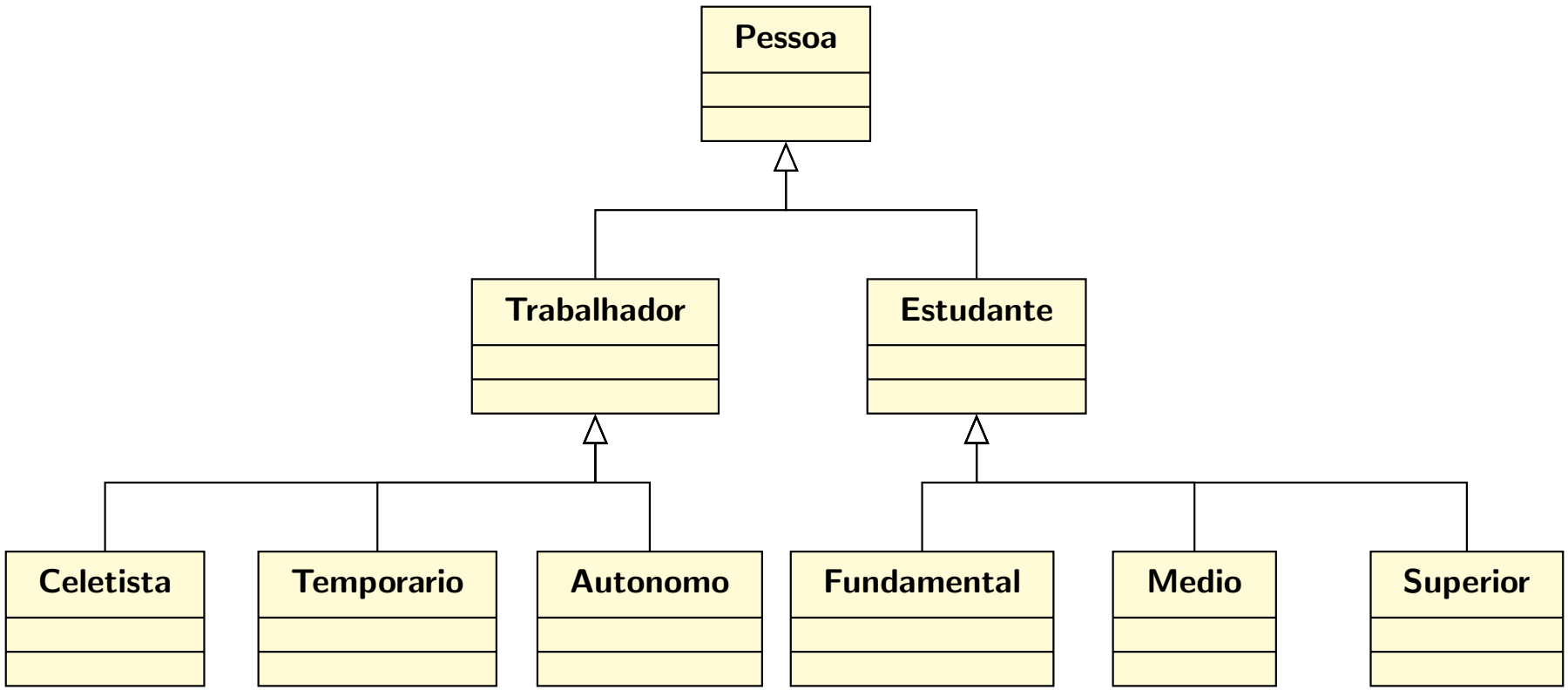
- A classe mais especializada é denominada *subclasse* e a mais genérica, *superclasse*:



- Mecanismo de herança como característica fundamental
  - Atributos e métodos da superclasse são herdados pela subclasse, exceto aqueles com modificadores *private*
  - Construtores não são herdados (mas é possível a subclasse ativar um método construtor da superclasse)

Especialização de classes (III)

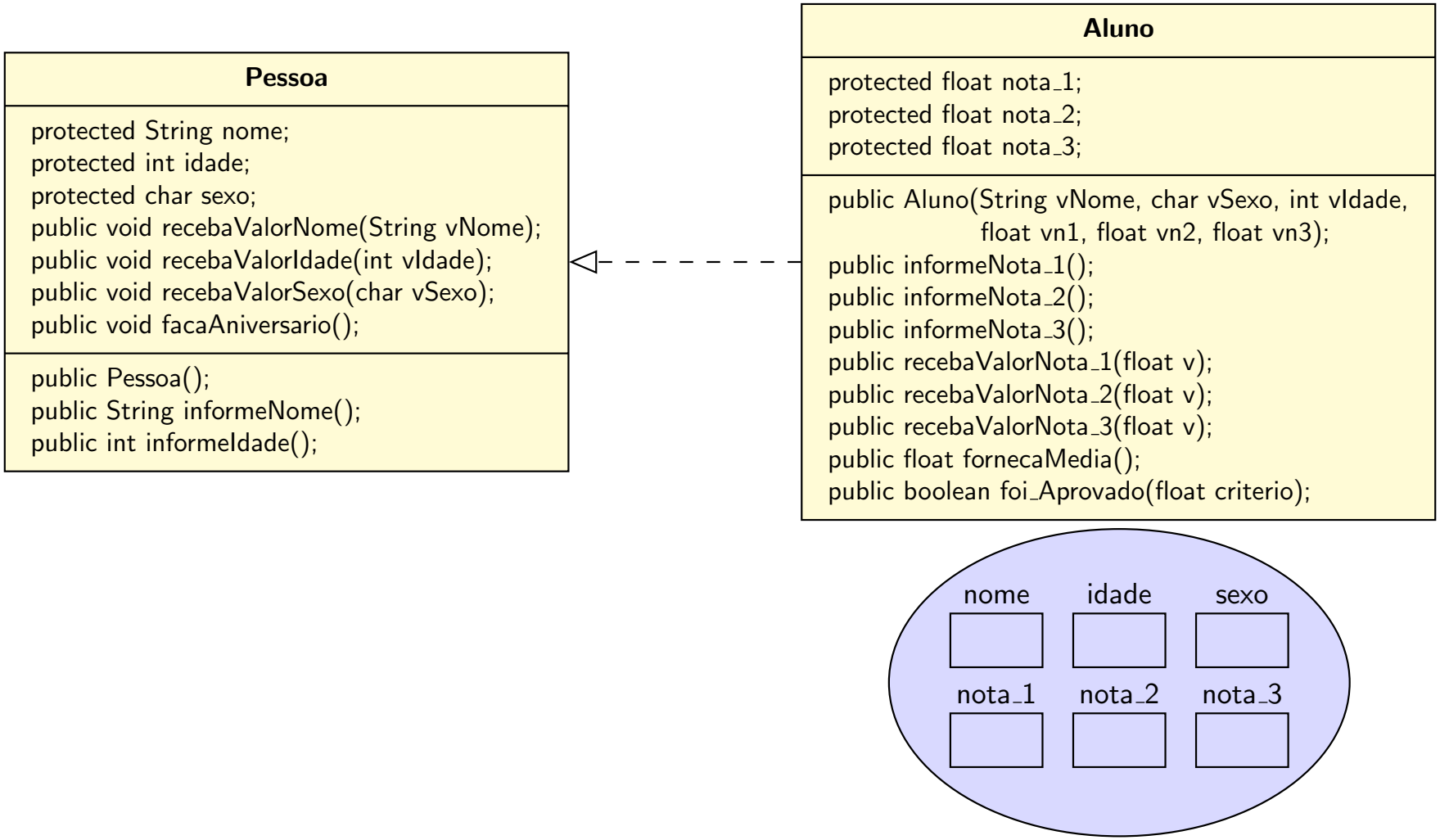
- Uma classe derivada pode ser especializada em outras classes:



Implementação de especialização – **extends**

- Em Java, a especialização é feita com uso da palavra-chave **extends**:

```
public class Aluno extends Pessoa { ... }
```



Implementação de especialização – exemplo

```
public class Aluno extends Pessoa {

 protected float nota_1;
 protected float nota_2;
 protected float nota_3;
 public Aluno(String vNome, char vSexo,
              int vIdade, float vn1,
              float vn2, float vn3) {
  //construtor da superclasse:
  super(vNome, vSexo, vIdade);
  if (vn1 < 0)       nota_1 = 0.0f;
  else if (vn1 > 10) nota_1 = 10.0f;
  else               nota_1 = vn1;
  if (vn2 < 0)       nota_2 = 0.0f;
  else if (vn2 > 10) nota_2 = 10.0f;
  else               nota_2 = vn2;
  if (vn3 < 0)       nota_3 = 0.0f;
  else if (vn3 > 10) nota_3 = 10.0f;
  else               nota_3 = vn3;
 }
 public float informeNota_1() {
  return nota_1;
 }
 public float informeNota_2() {
  return nota_2;
 }
 public float informeNota_3() {
  return nota_3;
 }
 public void recebaValorNota_1(float v) {
  if (v >= 0.0 && v <= 10.0) nota_1 = v;
 }
 public void recebaValorNota_2(float v) {
  if (v >= 0.0 && v <= 10.0) nota_2 = v;
 }
 public void recebaValorNota_3(float v) {
  if (v >= 0.0 && v <= 10.0) nota_3 = v;
 }
 public float fornecaMedia() {
  return (nota_1 + nota_2 + nota_3) / 3;
 }
 public boolean foi_Aprovado(float criterio
      ) {
  return this.fornecaMedia() >= criterio;
 }
}
```

Implementação de especialização – **super**

- No construtor da classe *Aluno*, usamos o comando:

```
super(vNome, vSexo, vIdade);
```

  - Ativa o construtor da superclasse (neste caso, *Pessoa*)
  - Inicializa os atributos herdados da superclasse
  - Se não for feito, Java coloca automaticamente uma chamada ao construtor padrão (sem nenhum parâmetro). Se não houver construtor padrão, neste caso, ocorrerá uma situação de erro

O mecanismo de herança (I)

- O processo de especialização de classe implica no mecanismo de herança
- Característica fundamental na programação orientada a objetos, pois possibilita a reutilização de código
  - A classe *Aluno*, por exemplo, reutiliza código da classe *Pessoa*
- Considerando um abstração de "aluno em dependência", com as mesmas avaliações de *Aluno*, e mais um conceito qualitativo de média:

Média final	*Conceito dependência*
Até 3,0	Reprovado
Maior que 3,0 até 6,0	Insuficiente
Maior que 6,0	Aprovado

O mecanismo de herança (II)

- Implementação

```
public class Aluno_Dependencia extends Aluno {
    public Aluno_Dependencia(String vNome, char vSexo, int vIdade,
                             float vn1, float vn2, float vn3) {
        super(vNome, vSexo, vIdade, vn1, vn2, vn3);
    }
    public String fornecaConcDependencia() {
        float media = this.fornecaMedia();
        //fornecaMedia foi herdado de Aluno
        if (media < 3.0) return "Reprovado";
        else
            if (media < 6.0) return "Insuficiente";
            else return "Aprovado";
    }
}
```

  - A classe *Aluno_Dependencia*, sendo subclasse de *Aluno*, terá os atributos *nome*, *sexo*, *idade*, *nota_1*, *nota_2* e *nota_3*

  - Toda instância da classe *Aluno_Dependencia* executa qualquer um dos métodos herdados da classe *Aluno*

O mecanismo de herança (III)

- Exemplo

```
//...
Aluno_Dependencia aluno_Dep;
aluno_Dep = new Aluno_Dependencia("Maria", 'F', 20, 10.0f, 8.0f, 9.5f);
String conceito = aluno_Dep.fornecaConcDependencia();
//...
```

  - A classe *Aluno_Dependencia* não adiciona atributo, ou seja, apresenta apenas os atributos herdados da classe *Aluno*
  - Seu construtor, portanto, necessita apenas ativar o construtor da classe *Aluno* para inicializar os atributos

- A herança é uma característica transitiva, isto é, ocorre por meio de múltiplos níveis de especialização
  - *Aluno* herdou de *Pessoa*; e *Aluno_Dependencia* herdou de *Aluno* e de *Pessoa*

O mecanismo de herança (IV)

- As características das classes superiores (também chamadas de classes ancestrais) são herdadas pela classe derivada
- De acordo com as regrada de Java, o modificador *private* possibilita a declaração de membros (atributos e métodos) que serão privativos da classe. Uma subclasse não herdará tais membros
- Por outro lado, tudo que for declarado com os modificadores *protected* ou *public* serão heradados pela subclasse

A Classe Object

- Em Java, toda classe tem a sua superclasse
- Se não for definida, Java automaticamente supõe a especialização de uma superclasse chamada *Object*
- A classe *Object* é parte da biblioteca padrão de Java (pacote `java.lang`, automaticamente importado), sendo a mais geral no topo da hierarquia, definindo o comportamento comum de todos os objetos
- A classe *Object* apresenta métodos, tais como:
  - `public Object()`: construtor padrão da classe
  - `public boolean equals(Object outro)`: informa se o executor e identificador *outro* referenciam a mesma instância
  - `public String toString()`: retorna o objeto representado na forma de um string

Reutilização de código

- Características herdadas não necessitam ser reescritas na subclasse
- Uma das principais características da modelagem orientada a objetos. O desenvolvimento de sistemas deve ter isto em mente
- A modelagem de resolução de um problema deve partir da definição de classes gerais. A especialização deve ser gradativa de forma a organizar a hierarquia
- Tais classes podem ser agrupadas em uma biblioteca para auxiliar no desenvolvimento de uma nova aplicação
- Ao escrever novas aplicações, pode-se levar sempre em consideração a definição de classes existentes, e proceder uma gradativa ampliação da biblioteca

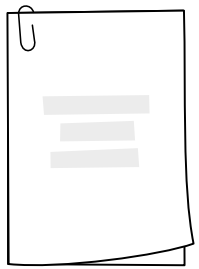
Reutilização de código e complexidade do problema

- Com uma boa biblioteca de classes, o índice de reutilização de código tende a aumentar a cada nova aplicação desenvolvida, e o custo de desenvolvimento tende a reduzir

- A especialização também pode representar uma abstração para melhor administração da complexidade de um problema. Exemplo:
  - Superclasse *Imovel* para todo imóvel (proprietário, valor, endereço, área, ...)
  - Subclasses de *Imovel*: *Urbano* (registro, IPTU, ...) e *Rural* (nome, ITR, área preservação, ...)
  - Subclasses de *Urbano*: *Casa* (nº cômodos, nº pavimentos, ...) e *Lote* (largura, profundidade, ...)
  - Subclasses de *Rural*: *Sítio* (caseiro, ...) e *Fazenda* (administrador, atividade, ...)

- Divisão do domínio do problema maior em domínios menores (problemas mais simples), facilitando a manipulação do problema como um todo

Exercício: Problema resolvido da seção 6.4



- Ler seção 6.4 sobre o Jogo de Dados
- Entender o problema e a modelagem, destacando a especialização realizada
- Testar o código:

  https://www.inf.ufsc.br/~alexandre.silva/courses/16s1/ine5603/codigos/jogoDados.java

Sobreposição de métodos (I)

- A **sobreposição** ou **redefinição** ou **overriding** consiste em uma subclasse requerer uma nova implementação de um método herdado

- A sobreposição de métodos está diretamente associada ao conceito de **polimorfismo** (será visto adiante)

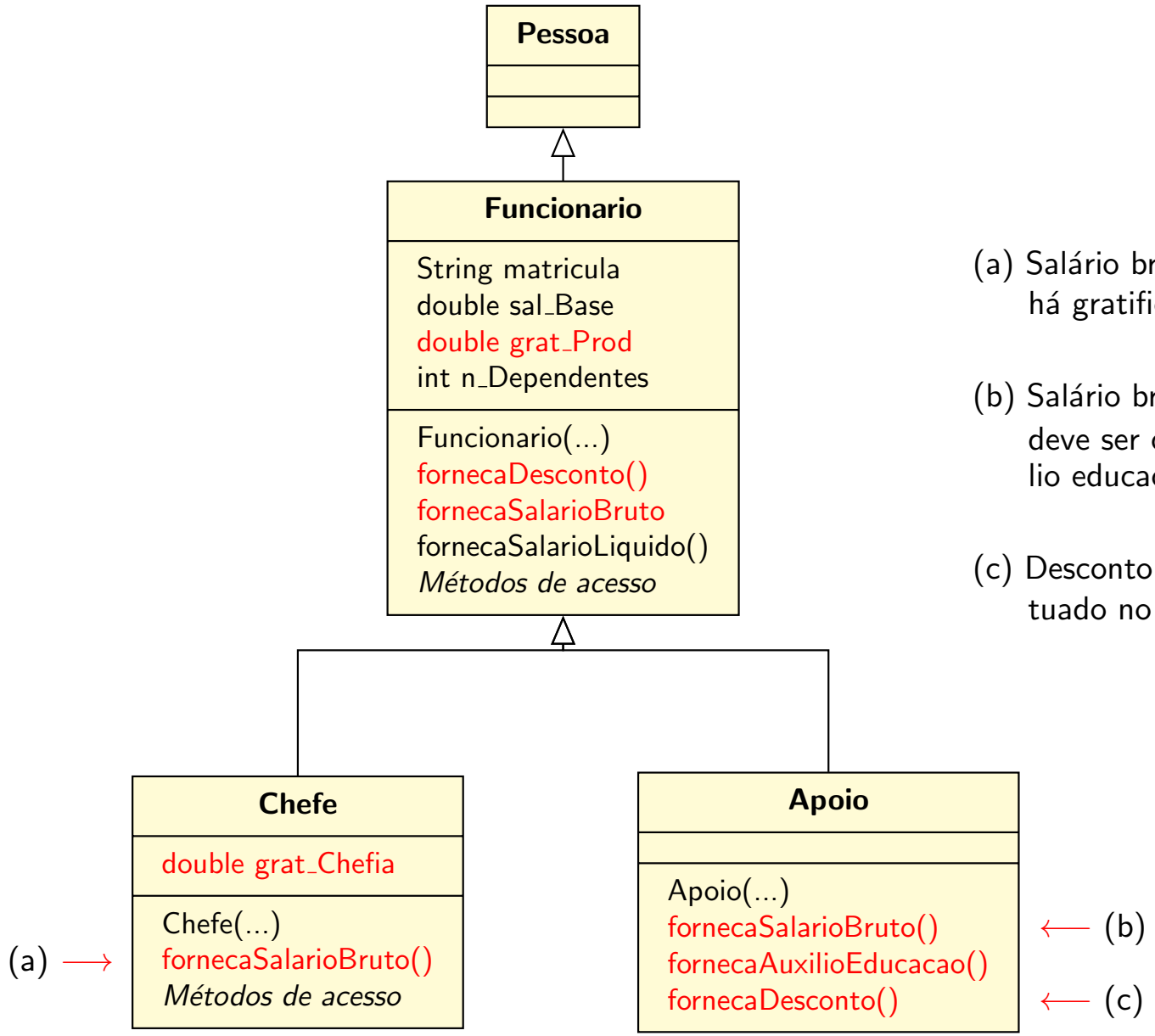
- Exemplo de problema:
  - *Processar a folha de pagamento dos funcionário de uma empresa, considerando que todo funcionário recebe salário base e gratificação de produtividade. Para os funcionários que exercem cargo de chefia, a empresa paga uma gratificação adicional. Considere um desconto sobre o salário bruto, para fins de imposto, conforme a tabela a seguir:*

Salário Bruto	**Alíquota** (%)	**Parcela a deduzir**
Até R$ 1000,00	isento	-
Acima de R$ 1000,00 e até R$ 1800,00	10	R$ 100,00
Acima de R$ 1800,00	25	R$ 370,00

Sobreposição de métodos (II)

- Exemplo de problema (cont...):
  - *Para funcionários (de apoio) com grau de instrução primário, a empresa para auxílio educação de R$ 60,00 por dependente, limitado a até cinco dependentes. Sobre tal auxílio, não incide qualquer desconto. Para cada funcionário, é preciso apresentar o seu demonstrativo salarial: nome, matrícula, salário-base, gratificação de produtividade, salário bruto, etc*
  - Pela descrição, a empresa tem tratamento diferenciado para o cálculo de salários (além dos funcionários comuns, há aqueles com cargo de chefia e outros de apoio)
  - Os funcionários da empresa podem ser modelados na classe `Funcionario` (subclasse de `Pessoa`), a ser especializada nas subclasses `Chefe` e `Apoio`

Sobreposição de métodos (III)



(a) Salário bruto diferente pois há gratificação específica

(b) Salário bruto diferente pois deve ser considerado o auxílio educação

(c) Desconto diferente não efetuado no auxílio educação

Sobreposição de métodos (IV)

- Há sobreposição de um método quando a correspondente tarefa deve ser executada de maneira diferente na subclasse
- É nessário que o método redefinido tenha a mesma assinatura do que está sendo sobreposto
- Dois métodos têm mesma assinatura se tiverem o mesmo nome, quantidade de parâmetros, sendo, pela ordem, de mesmos tipos
- Além de sobreposição, Java possibilita sobrecarga (*overload*) de métodos. Neste caso, ocorrem duas ou mais assinaturas diferentes (veremos a seguir)

Sobreposição de métodos (V)

- O método da superclasse pode ser acessado com `super`
- Na classe `Funcionario`, o método `FornecaSalarioBruto()` é:

```
public double fornecaSalarioBruto() {
    return sal_Base + grat_Prod;
}
```

- Na classe `Chefe`, o método `FornecaSalarioBruto()` é:

```
public double fornecaSalarioBruto() {
    return sal_Base + grat_Prod + grat_Chefia;
}
```

- Logo, na classe `Chefe`, o método `FornecaSalarioBruto()` também pode ser:

```
public double fornecaSalarioBruto() {
    return super.fornecaSalarioBruto() +
        grat_Chefia;
}
```

Sobrecarga de métodos (I)

- Quando há uma diferente implementação na subclasse, temos **sobreposição** (*overriding*) de métodos $\Rightarrow$ **mesma assinatura e há substituição!**

- É possível, em Java, a **sobrecarga** (*overloading*) de métodos $\Rightarrow$ **diferentes assinaturas e não há substituição!**

  - O uso de dois ou mais construtores de uma classe é um exemplo de sobrecarga de métodos

  - Possibilita execução com parâmetros de tipos diferentes ou em quantidades diferentes de parâmetros

Sobrecarga de métodos (II)

- Mais um exemplo:
  maior valor entre dois ou três *double*, ou entre dois *char*

```
public class Principal {

  public static void main(String[] args) {
    Principal p = new Principal();
    System.out.println(p.maior(3.4, 7.0, 5.6));
    //imprime o valor 7.0
    System.out.println(p.maior('g', 'H'));
    //imprime o caracter g
  }

  public double maior(double a, double b) {
    if (a > b) return a;
    else       return b;
  }

  public double maior(double a, double b, double c) {
    return this.maior(a, this.maior(b,c));
  }

  public char maior(char x, char y) {
    if (x > y) return x;
    else       return y;
  }
}
```

Compatibilidade de endereços (subtipagem)

- Um identificador de objeto pode se referir (conter o endereço) a uma instância de qualquer classe descendente. Exemplo:

```
Funcionario func;
func = new Apoio("Maria", "1234-x", 500.0, 120.0, 3);
//...
func = new Chefe("Paula", "115-8", 2500.0, 430.0, 3, 560.0);
//...
func = new Funcionario("Ze", "9175-2", 800.0, 90.0, 3);
```

- Uma variável que identifica um objeto é também denominada polimórfica, neste sentido

- Uso de métodos específicos com conversão explícita de tipos (dentro de uma mesma árvore de classes). Exemplo:

```
Funcionario func;
func = new Chefe("Paula", "115-8", 2500.0, 430.0, 3, 560.0);
((Chefe)func).recebaValorGrat_Chefia(300.0);
```

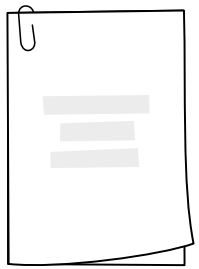
Operador `instanceof`

- O `instanceof` é utilizado quando há necessidade de saber se um identificador está referindo a uma instância de determinada classe
- Esse operador, aplicado a um identificar e a uma classe, forma uma expressão tipo booleana. Por exemplo, se a expressão

```
umFuncionario istanceof Chefe
```

apresenta o valor *true*, então o identificador `umFuncionario` está referindo uma instância que é um `Chefe`

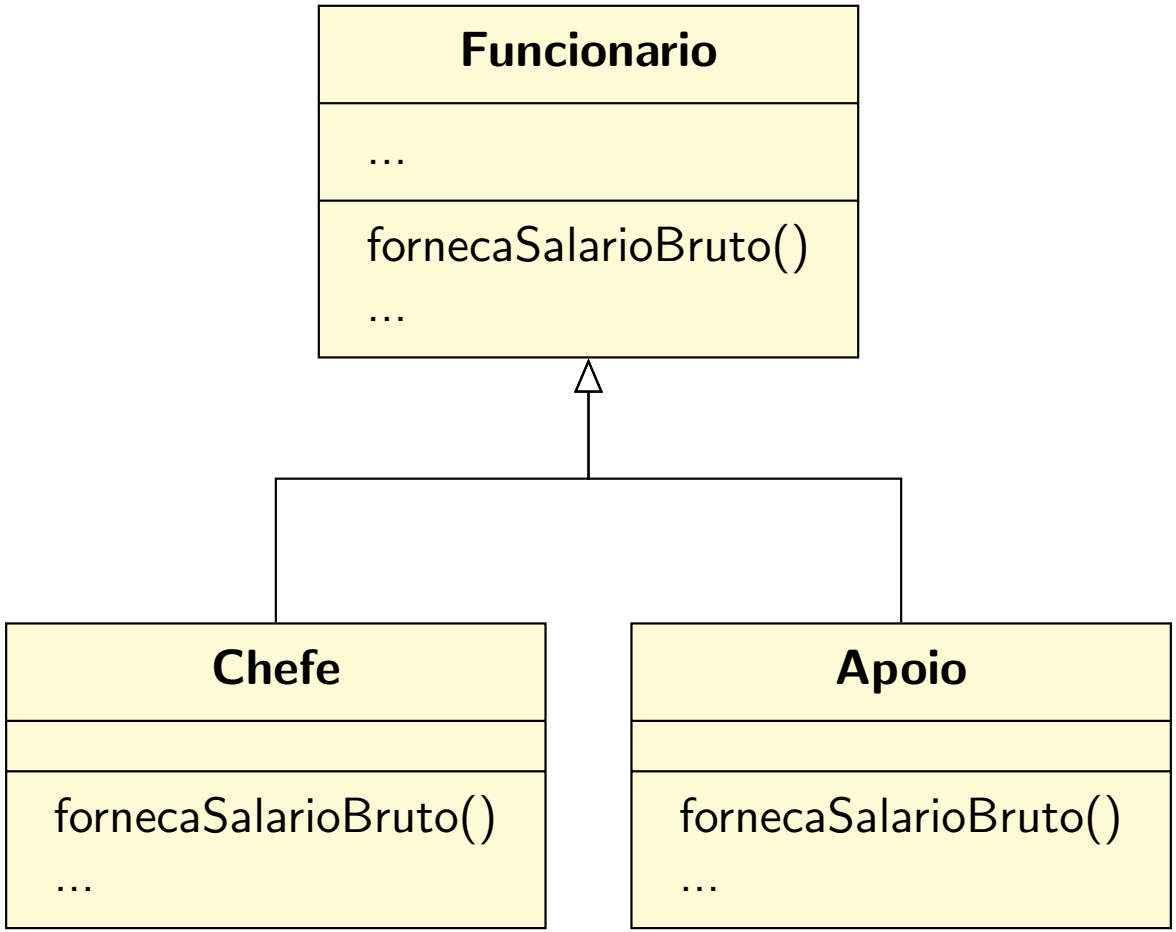
Exercício: implementação da seção 6.5



- Analise e execute a implementação das classes `Pessoa`, `Funcionario`, `Apoio`, `Chefe`, `Interface` e `Principal` em:
  - https://www.inf.ufsc.br/~alexandre.silva/courses/15s2/ine5603/codigos/funcionarios/

Polimorfismo (I)

- Polimorfismo de método



Polimorfismo (II)

- **Sobreposição de método** diretamente associada ao polimorfismo de método
- Método automaticamente selecionado em função da classe da instância executora
- **Polimorfismo de método**:

```
Funcionario umF;
double sb;
umF = new Funcionario("Tel", "11-5", 500.0, 30.0,
    3);
sb = umF.fornecaSalarioBruto();
// ...
umF = new Apoio("Pedro", "28-8", 450.0, 75.0, 3);
sb = umF.fornecaSalarioBruto();
```

Polimorfismo (III)

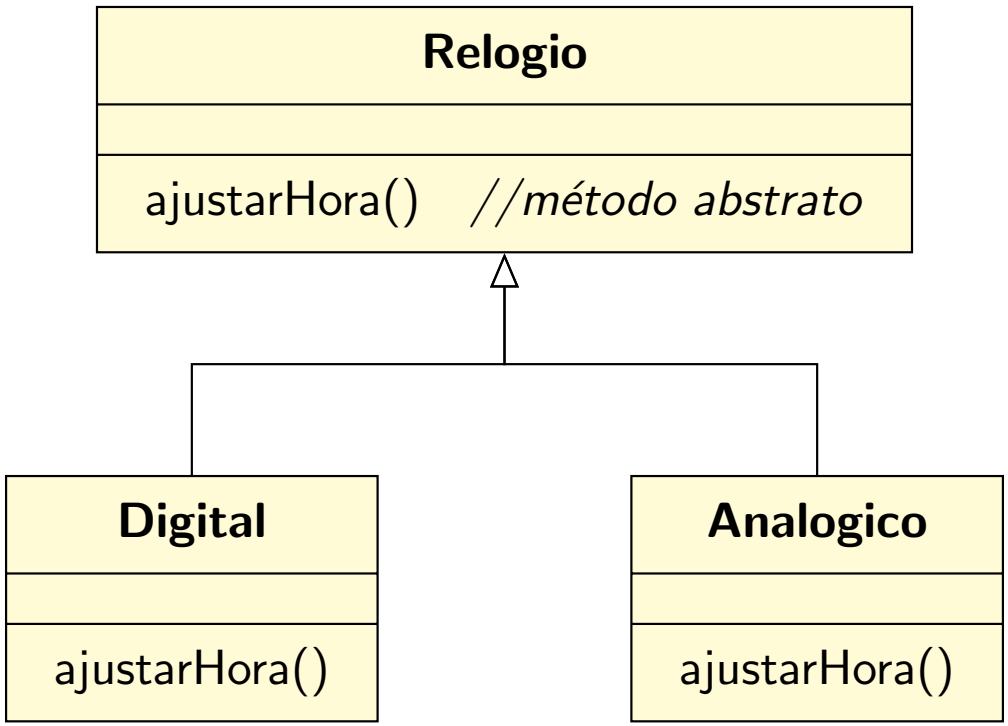
- **Variáveis polimórficas** (identificador que assume várias formas):

```
Pessoa p;
p = new Pessoa("Isaias", 'M', 50);
// ...
p = new Funcionario("Pedro", "28-8", 450.0, 75.0,
    3);
```

  - `Pessoa` é superclasse de `Funcionario`
  - Inicialmente o identificador `p` é endereço de uma instância da classe `Pessoa`
  - Em seguida, o identificador `p` passa a ser endereço de uma instância da classe `Funcionario`, que apresenta outra forma

Polimorfismo (IV)

- Exemplo do ajuste de relógio
  - Tarefa realizada de formas diferentes dependendo do tipo de relógio
  - O método na superclasse, de fato, não tem implementação, sendo denominado de **método abstrato**



Classes abstratas (I)

- Classes que apenas organizam características comuns, não existindo nenhuma instância da mesma no processo de resolução do problema
- Na modelagem, é possível que sejam definidos um ou mais métodos nos quais não cabe uma implementação
- Um método que não apresenta implementação é denominado de **Método Abstrato** e a respectiva classe é denominada **Classe Abstrata**
- Classe Abstrata é definida como sendo uma classe não instanciável, que apresenta pelo menos um método para o qual não existe implementação (embora possa especificar atributos e métodos concretos)

Classes abstratas (II)

- Exemplo de Classe Abstrata:
  - Imposto pessoa física:

Faixa renda líquida anual	Alíquota (%)	Parcela a deduzir
Até R$ 10000,00	Isento	-
De R$ 10000,00 a R$ 20000,00	10	R$ 1000,00
Acima de R$ 20000,00	25	R$ 4000,00

    - ⋆ A renda líquida anual é calculada a partir da renda bruta anual, subtraindo-se os abatimentes
    - ⋆ Consideramos que, para pessoa física, é permitido abater os gastos com saúde (até um limite de R$ 5000,00) e gastos com instrução (até um limite de R$ 2500,00) e também R$ 1500,00 por dependente (limitado ao máximo de quatro dependentes)

Classes abstratas (III)

- Exemplo de Classe Abstrata (*cont...*):
  - Imposto pessoa jurídica:

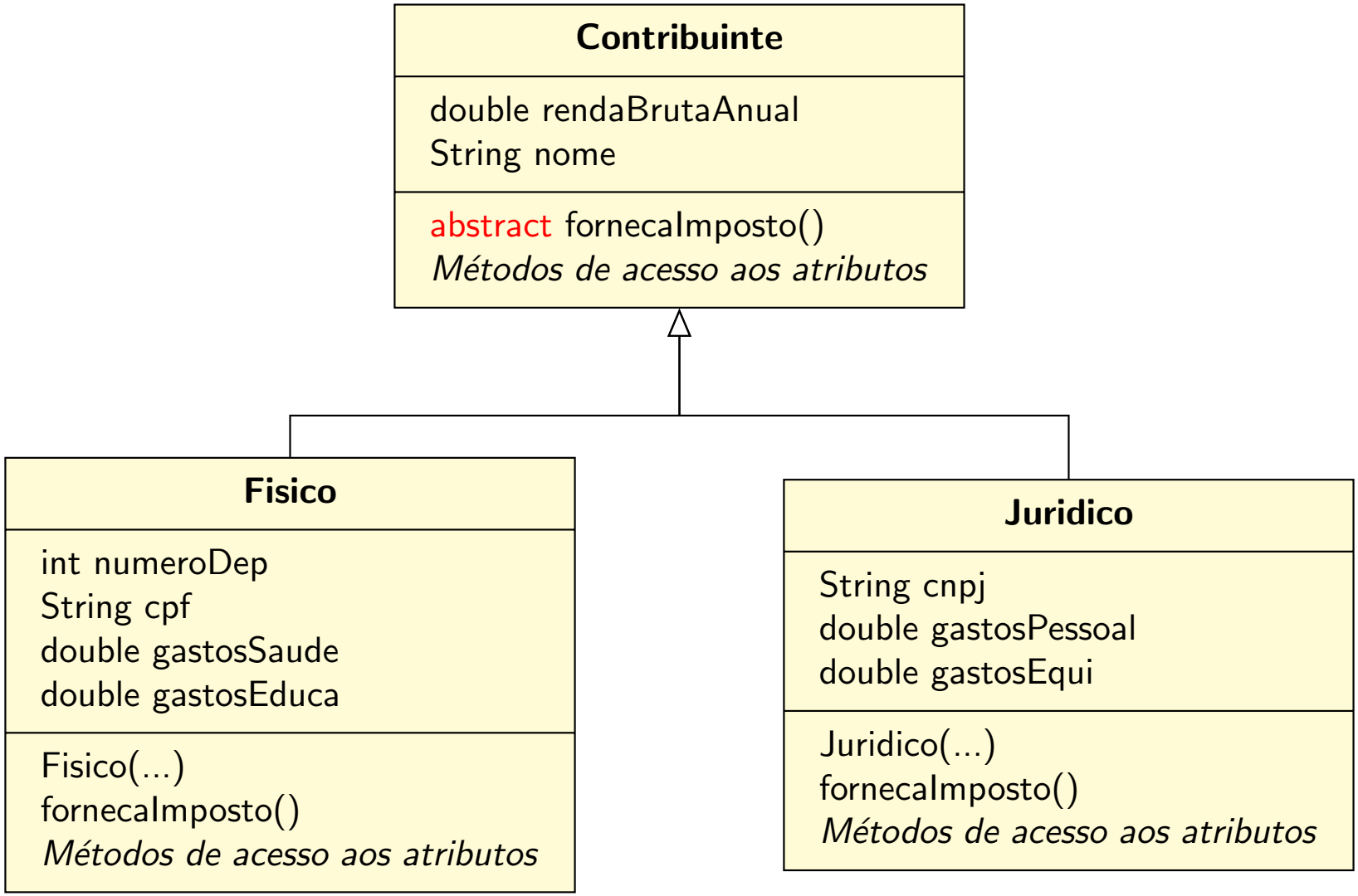
Faixa renda líquida anual	Alíquota (%)	Parcela a deduzir
Até R$ 100000,00	5	-
De R$ 100000,00 a R$ 200000,00	10	R$ 5000,00
Acima de R$ 200000,00	30	R$ 45000,00

    - ★ Consideramos que, para pessoa jurídica, a renda líquida anual é calculada a partir da renda bruta anual, subtraindo-se os abatimentes

- `Fisico` e `Juridico` são subclasses de `Contribuinte`, mas esta não implementa o método `fornecaImposto()` pois seu cálculo é diferente a cada tipo de contribuinte (não há uma forma de cálculo geral)

Classes abstratas (IV)

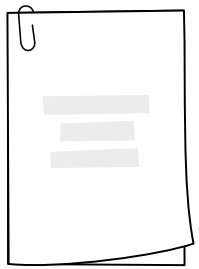
- Modelagem



Classes abstratas (V)

- A classe `Contribuinte` agrega as características comuns das classes `Fisio` e `Juridico`, facilitando a manipulação da complexidade e permitindo a reutilização de código
- O método `fornecaImposto` deve ser implementado nas subclasses concretas, pois trata-se de um método abstrato herdado
- A classe final de um ramo da especialização deve ser concreta (não faz sentido uma classe abstrata como uma folha na hierarquia de classes)
- Nada impede que, a partir de uma classe abstrata, derive-se outra classe abastrata

Exercício: implementação da seção 6.8



- Analise, execute e adicione testes à implementação das classes `Contribuinte`, `Fisico` e `Juridico` em:
  - https://www.inf.ufsc.br/~alexandre.silva/courses/15s2/ine5603/codigos/contribuinte/

Exercício: implementação da seção 6.8 (I)

```
public abstract class Contribuinte {
    //observe a necessidade de declaracao como 'abstract'

    protected String nome;
    protected double rendaBrutaAnual;

    public double informeRendaBrutaAnual() {
        return rendaBrutaAnual;
    }

    public abstract double fornecaImposto();
    //metodo fornecaImposto e' abstrato e nao tem implementacao
}
```

Exercício: implementação da seção 6.8 (II)

```
public class Fisico extends Contribuinte {
  protected String cpf;
  protected int numeroDep;
  protected double gastosSaude;
  protected double gastosEduca;
  public Fisico(String vNome, float vRBAnual, String vCpf, int
      vNumDep, double vGS, double vGE) {
    //inicializacoes...
  }
  public double fornecaImposto() {
    //implementacao do metodo abstrato herdado
    double abatSaude, abatEduca, abatDep, liquida;
    abatSaude = gastosSaude;
    if (abatSaude > 5000.00)       abatSaude = 5000.00;
    abatEduca = gastosEduca;
    if (abatEduca > 2500.00)       abatEduca = 2500.00;
    abatDep = numeroDep * 1500.00;
    if (abatDep > 6000)            abatDep   = 6000;
    liquida = rendaBrutaAnual - abatSaude - abatEduca - abatDep;
    if (liquida < 10000.00)        return 0.0;
    else if (liquida <= 20000.00) return liquida * 0.1 - 1000.00;
    else                           return liquida * 0.25 - 4000.00;
  }
}
```

Exercício: implementação da seção 6.8 (III)

```
public class Juridico extends Contribuinte {
    protected String cnpj;
    protected double gastosPessoal;
    protected double gastosEqui;
    public Juridico(String vNome, float vRBAnual, String vCnpj,
        double vGP, double vGE) {
      //inicializacoes...
    }
    public double fornecaImposto() {
        //implementacao do metodo abstrato herdado
        double liquida = rendaBrutaAnual - gastosPessoal -
            gastosEqui;
        if (liquida < 0)
            return 0;
        else
            if (liquida <= 100000.00)
                return liquida * 0.05;
            else
                if (liquida <= 200000.00)
                    return liquida * 0.1 - 5000.00;
                else
                    return liquida * 0.3 - 45000.00;
    }
}
```