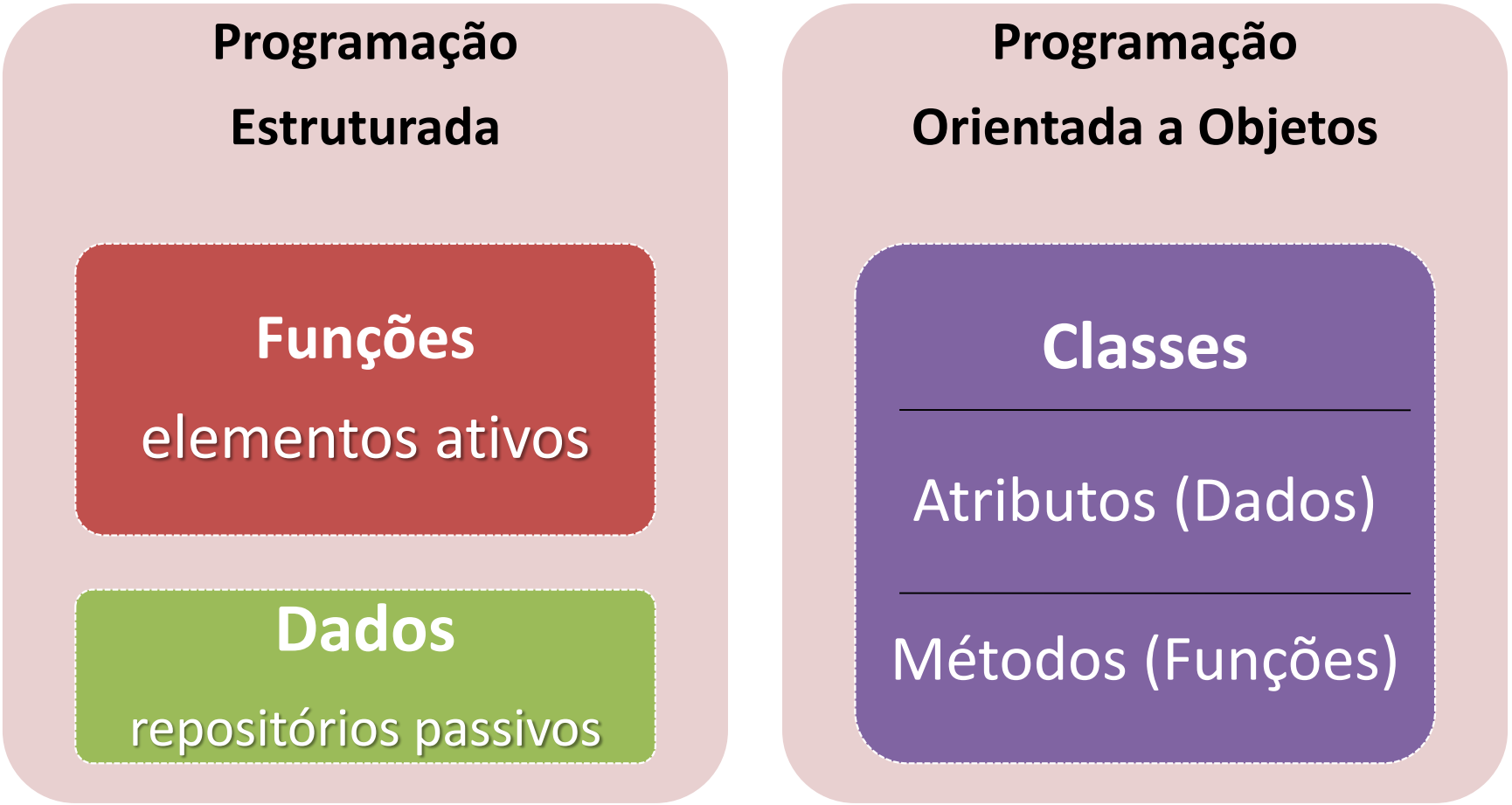
Linguagem de Programação

Orientada a Objetos

Profa. Joyce Miranda

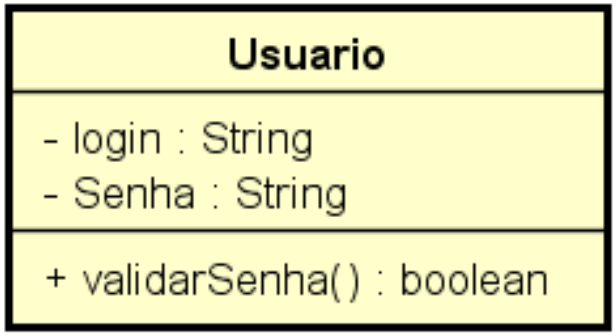
Apresentação do Módulo

- **Visão Geral**



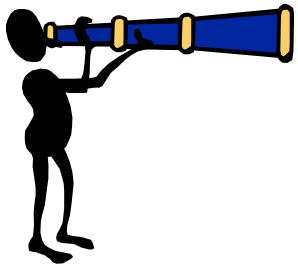
Introdução

- Programação **Orientada a Objetos** (POO)
  - O código é organizado sob a ótica de classes que definem atributos (dados) e métodos (comportamento) que são comuns a objetos de um mesmo tipo.
    - Permite maior reaproveitamento de código e facilidade de manutenção.



Introdução

- Paradigma **Orientado a Objetos** (OO)
  - Baseia-se na *abstração.*



CARRO

COR: VERDE
QTDE RODAS: 4
QTDE PASSAGEIROS: 1
NUMERO: 1
AÇÃO: CORRE, ACELERA

INDIVÍDUO **REALIDADE** **ESTRUTURA**

Conceitos - Objetos

- Um objeto é um conceito, uma abstração, algo com limites e significados nítidos em relação ao domínio de uma aplicação.
- Para cada sistema devem ser identificados objetos de acordo com o contexto no qual está inserido e de acordo com as funcionalidades desejadas.

Domínio Acadêmico

Domínio Locadora

Conceitos - Objetos

- Os objetos podem ser agrupados de acordo com as suas semelhanças.

Domínio Acadêmico

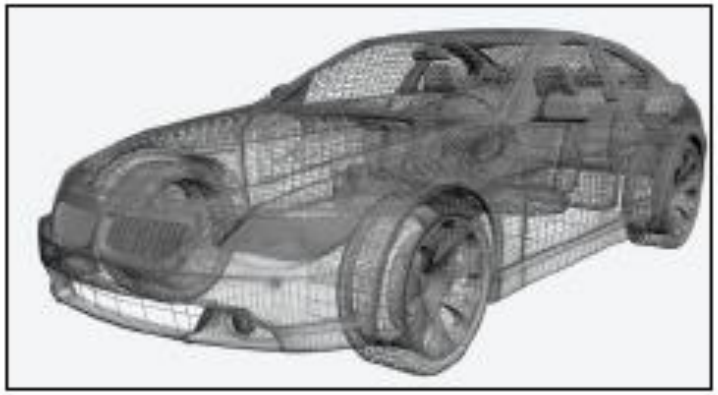
Professor Fulano
Professor Ciclano
Professor Beltrano

Curso Informática
Curso Edificações
Curso Turismo

Aluno Alfa
Aluno Beta
Aluno Gama

Conceitos - Classes

- Objetos podem ser agrupados em classes
- Uma classe é um modelo que **define** os atributos e os métodos comuns a todos os objetos do mesmo tipo.



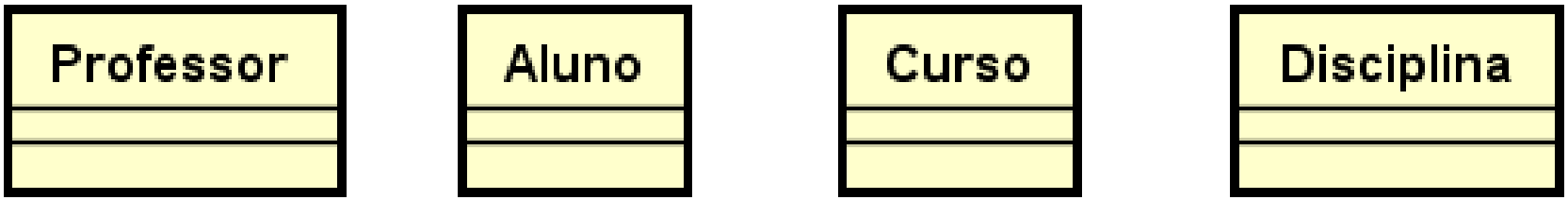
Classe

Objeto

Conceitos - Classes

- Exemplos de classes por domínio

Domínio Acadêmico



Conceitos - Classes

- Exemplos de classes por domínio

Domínio Locadora

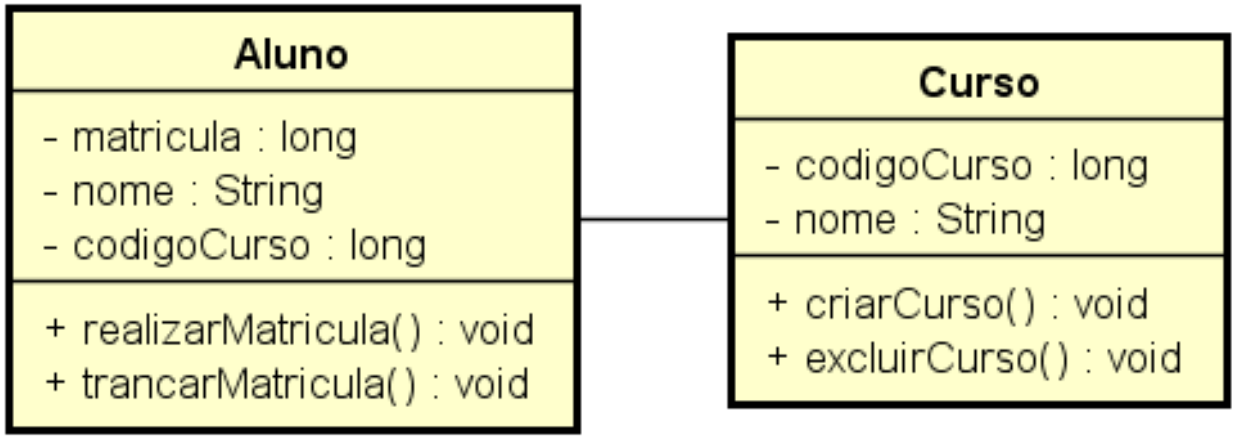
?

?

?

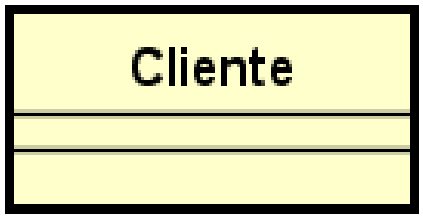
Conceitos - Classes

- Uma classe é a descrição de um grupo de objetos com ***propriedades semelhantes*** (atributos), ***mesmos comportamentos*** (métodos) e ***mesmos relacionamentos*** com outros objetos de outras classes.



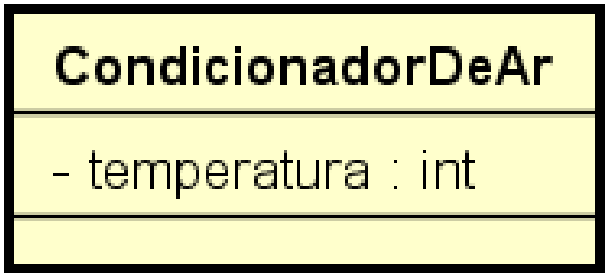
Conceitos - Classes

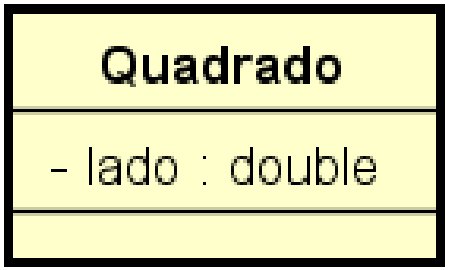
- Atributos
  - São as características que vão ajudar a representar um objeto.
  - Definem a estrutura de dados que vai representar a classe.
  - Quais atributos poderiam ser definidos para as classes abaixo?



Conceitos - Classes

- Métodos
  - São as tarefas/ações que o objeto pode realizar.
  - Quais métodos poderiam ser definidos para as classes abaixo?





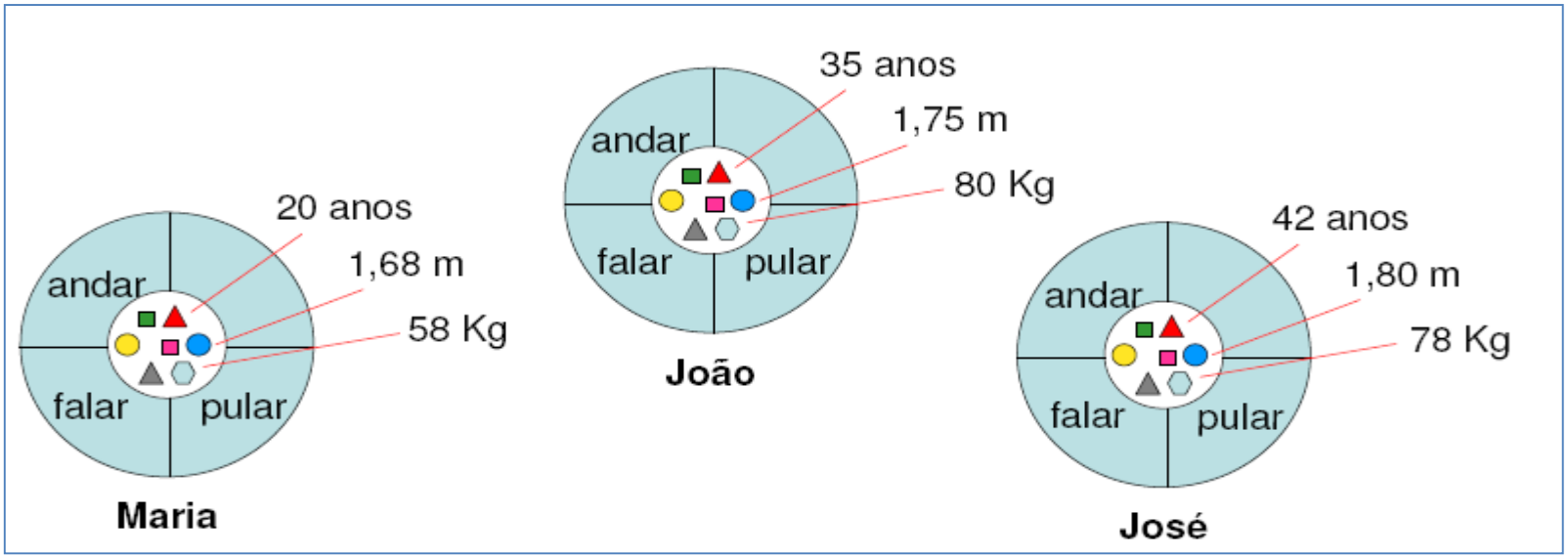
Conceitos - Classes

- Praticando
  - **Proponha pelo menos três classes, com seus respectivos atributos, métodos e relacionamentos para uma aplicação que esteja inserida no domínio de Gerenciamento Bancário.**

Domínio Bancário

Conceitos - Objetos

- **Instância**
  - Um sistema pode conter um ou mais objetos ativos.
  - Cada objeto ativo no sistema em particular é chamado de **instância**.
  - As diferentes instâncias possuem seu próprio estado.



Objetos na Prática

- Um objeto, nada mais é do que uma instância de um tipo de dado específico (classe).

```
Carro meuCarro = new Carro( );
```

Declaração da variável que vai guardar uma **referência para um objeto** do tipo Carro.

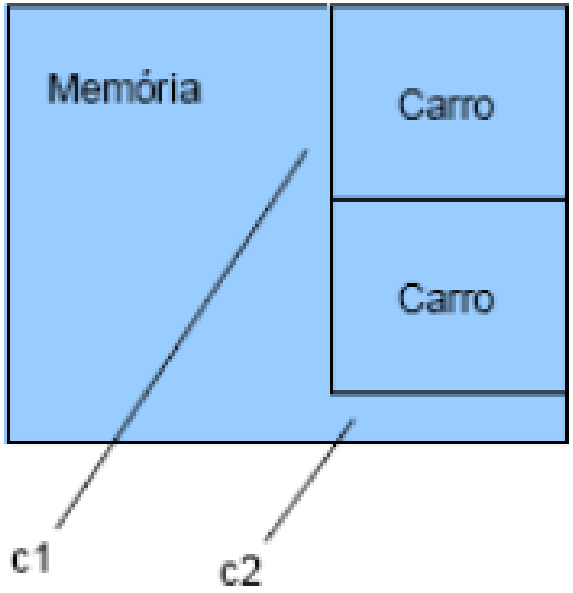
Operador **new** instancia o objeto.

Construtor da classe Carro

Objetos na Prática

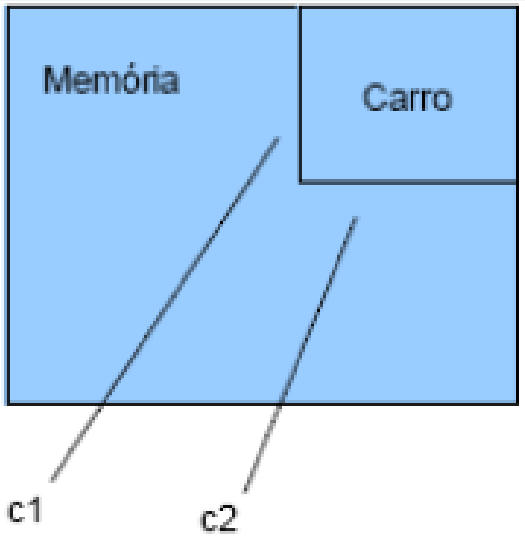
- As variáveis não guardam os objetos, mas sim uma **referência para a área de memória** onde os objetos estão alocados.

```
Carro c1 = new Carro( );
Carro c2 = new Carro( );
```



Objetos na Prática

- Imagine, agora, duas variáveis diferentes, *c1 e c2, ambas* referenciando o mesmo objeto. Teríamos, agora, um cenário assim:




```
Carro c1 = new Carro( );
Carro c2 = c1;
```

Conceitos - Classes

- Considere um sistema para gerenciar um banco.

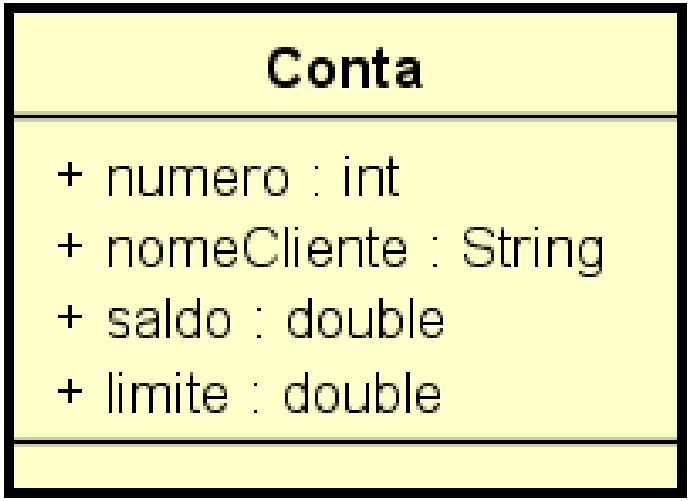
- Entidade Fundamental: CONTA

Conceitos - Classes

- O que toda conta deve possuir?
  - Número da conta
  - Nome do titular da conta
  - Saldo
  - Limite

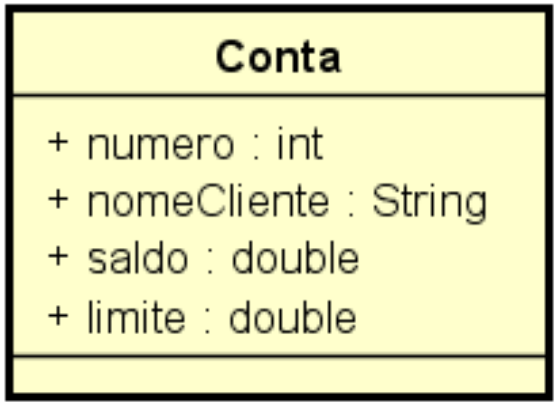
Conceitos - Classes

- Projeto de Conta
  - Definição da Classe
  - Identificação dos Atributos



Conceitos - Classes

- Classes na Prática




```
public class Conta {

    public int numero;
    public String nomeCliente;
    public double saldo;
    public double limite;

}
```

Conceitos - Classes

- Usando a Classe
  - Criar uma classe de execução que implemente o método *main*
  - Instanciar -> criar objeto

```
public class ExecConta {

    public static void main(String args[]){

        //instanciando ->criando objeto
        Conta minhaConta = new Conta();

    }

}
```

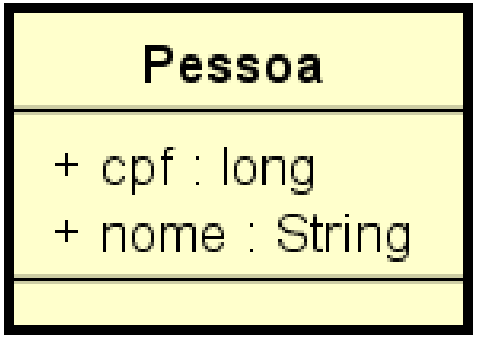
Conceitos - Classes

```
public class ExecConta {

    public static void main(String args[]){

        //instanciando ->criando objeto
        Conta minhaConta = new Conta();

        minhaConta.numero = 0001;
        minhaConta.nomeCliente = "Fulano de Tal";
        minhaConta.saldo  = 1000.00;
        minhaConta.limite = 300.00;

        System.out.println("O saldo da conta do cliente: " +
                           minhaConta.nomeCliente +
                           " é :" +
                           minhaConta.saldo);
    }

}
```

- Utiliza-se o ponto (.) para acessar os atributos e métodos de um objeto.

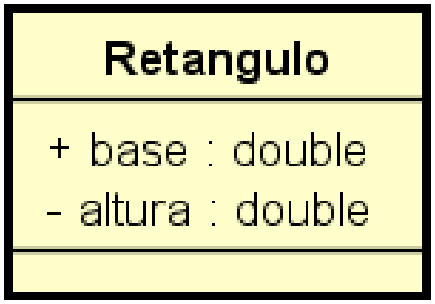
Classes na Prática

- Implemente a classe Pessoa.
- Crie uma classe de execução onde deverá ser criado um objeto do tipo Pessoa.
- Defina valores paras os atributos do objeto criado.
- Imprima uma mensagem com os valores dos atributos.



Classes na Prática

- Implemente a classe Retangulo.
- Crie uma classe de execução para criar o objeto do tipo Retangulo e defina valores para seus atributos.
- Os valores dos atributos do objeto deverão ser definidos dinamicamente por meio da interação com o usuário.



Classes na Prática

Construtores

- Quando usamos a palavra chave **new**, estamos construindo um objeto.
- Sempre quando o **new** é chamado, ele executa o construtor da classe.
  - Fazem a função de iniciação do objeto criado.

```
Conta minhaConta = new Conta();
```

Chamada do Construtor

Classes na Prática

- Construtores
  - O construtor da classe é um bloco declarado com o mesmo nome que a classe.
  - Se nenhum construtor for declarado, um construtor *default* será criado.

```
Conta minhaConta = new Conta();
```

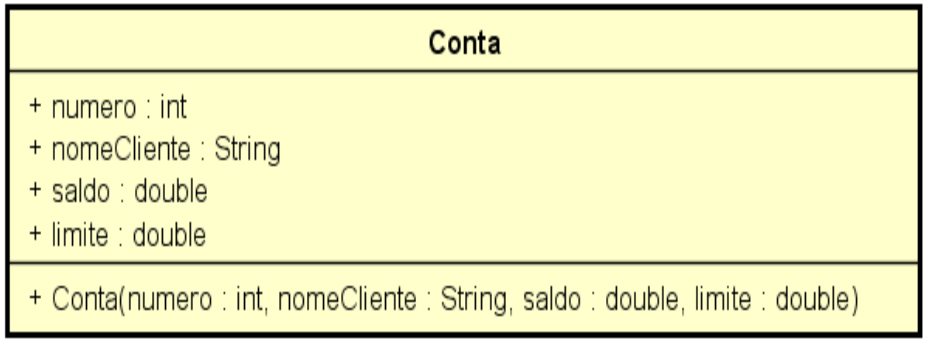
Contrutores

```
public class Conta {

    public int numero;
    public String nomeCliente;
    public double saldo;
    public double limite;

    public Conta(){}

}
```

Declaração implícita do Construtor

```
Conta minhaConta = new Conta();
```

Contrutores




```
public class Conta {

    public int numero;
    public String nomeCliente;
    public double saldo;
    public double limite;

    public Conta(int numero, String nomeCliente, double saldo, double limite){
        this.numero = numero;
        this.nomeCliente = nomeCliente;
        this.saldo = saldo;
        this.limite = limite;
    }
}
```

Alteração do Construtor

```
Conta minhaConta = new Conta(0001, "Fulano", 1000.00, 600.00);
```

Classes na Prática

- Construtores

```
public class ExecConta {

    public static void main(String args[]){

        Conta minhaConta = new Conta(0001, "Fulano", 1000.00, 600.00);

        System.out.println("Nome do Cliente: " + minhaConta.nomeCliente);
        System.out.println("Saldo atual: " + minhaConta.saldo);
    }

}
```

```
Nome do Cliente: Fulano
Saldo atual: 1000.0
```

Pratique!

- Construtor
  - Classe Modelo + Classe de Execução

Pessoa
+ cpf : long + nome : String
+ Pessoa(cpf : long, nome : String)

Retangulo
+ base : double + altura : double
+ Retangulo(base : double, altura : double)

Classes na Prática

- Métodos
  - A utilidade dos métodos é a de separar uma determinada função em pedaços menores.

```
<tipo de retorno> <nome do método>( [lista dos atributos] ) {
        // implementação do método
}
```

  - tipo de retorno
    - Pode ser um tipo primitivo ou um tipo de um classe.
    - Caso o método não retorne nada, ele deve ser **void**.
  - A lista de atributos não precisa ser informada se não há passagem de argumentos.
    - Caso haja, os argumentos devem ser informados com seu tipo e nome, separados por vírgula se houver mais de um.

Classes na Prática

- Métodos

```
void somaValores(int a, int b){
    int soma;
    soma = a + b;
    System.out.println("A soma é: " + soma);
}
```

Classes na Prática

- Métodos
  - Retorno dos Métodos
    - A palavra reservada ***return*** causa o retorno do método.
    - Quando os métodos são declarados com o tipo de retorno **void**, então o método não pode e nem deve retornar nada.
    - Os métodos que retornam algum valor, devem retornar dados do tipo de retorno declarado, ou de tipos compatíveis.

Classes na Prática

- Métodos
  - Retorno dos Métodos

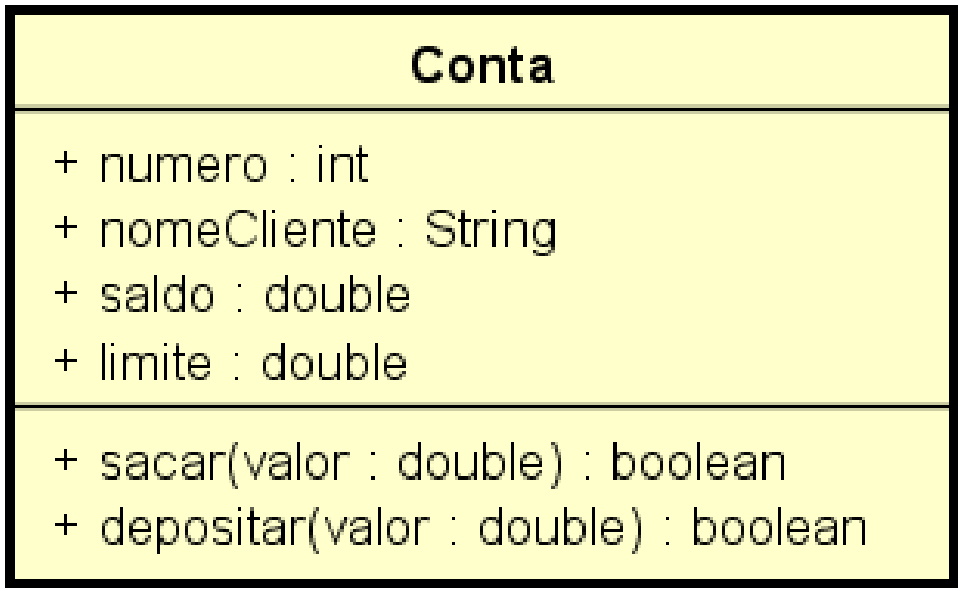
```
int somaValores(int a, int b){
    int soma;
    soma = a + b;
    return soma;
}
```

Conceitos - Classes

- Que ações podem ser feitas sobre a Conta?
  - Realizar o saque de um valor
  - Depositar um valor
  - Consultar/Imprimir informações da conta

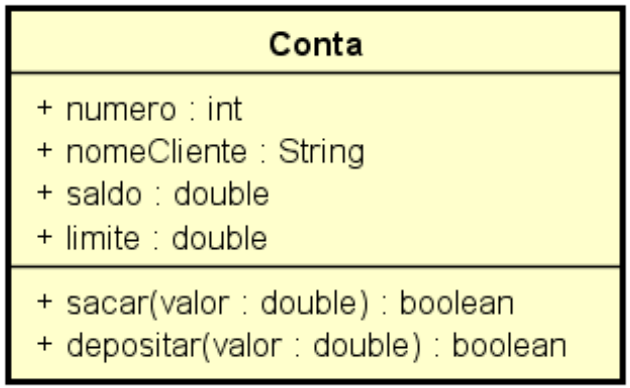
Conceitos - Classes

- Projeto de Conta
  - Identificação dos Métodos



Conceitos - Classes

- Métodos




```
public class Conta {

    public int numero;
    public String nomeCliente;
    public double saldo;
    public double limite;

    public boolean sacarValor(double valor){
        if(this.saldo < valor){
            return false;
        }else{
            this.saldo = this.saldo - valor;
            return true;
        }

    }
}
```

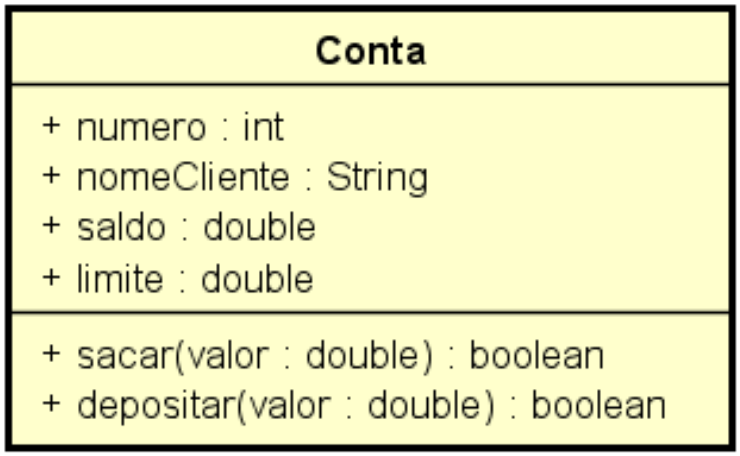
Conceitos - Classes

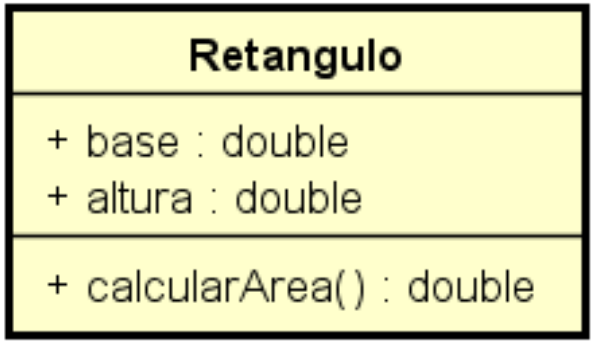
- Chamando Métodos

```
public class ExecConta {

    public static void main(String args[]){

        Conta minhaConta = new Conta();

        minhaConta.saldo = 1000.00;
        boolean consegui = minhaConta.sacarDinheiro(500.00) ;
        if(consegui){
            System.out.println("Saque realizado com sucesso!");
        }else{
            System.out.println("Saque não realizado!");
        }
    }

}
```

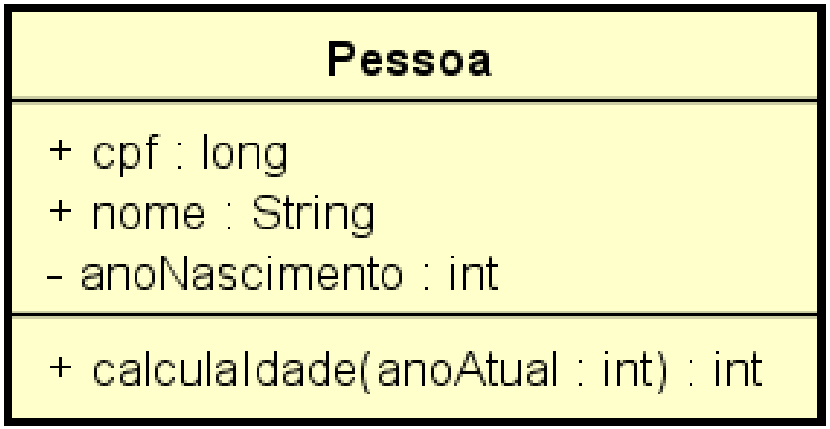
Pratique!

- Método
  - Classe Modelo e Classe de Execução





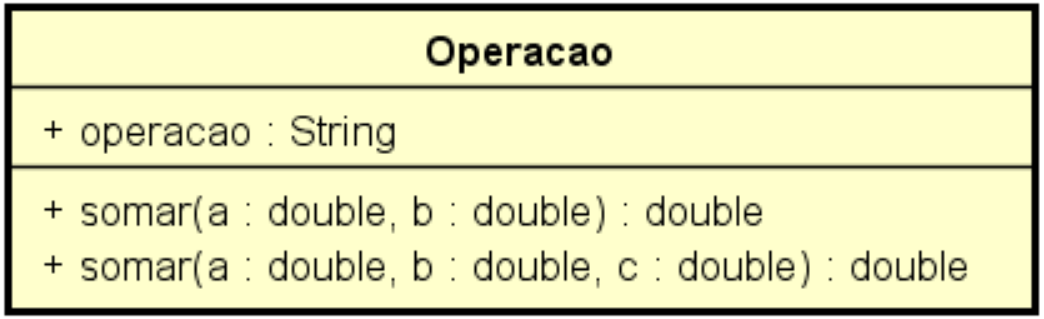
Pratique!




```
//recuperando ano atual

Calendar c = Calendar.getInstance();

int anoAtual = c.get(Calendar.YEAR);
```

Classes na Prática

- Sobrecarga – *Overload*
  - Ocorre quando mais de um método com o mesmo nome é implementado.
  - Pode se dar pela diferenciação do **tipo de retorno** e/ou dos **argumentos do método**.



Classes na Prática

- Sobrecarga de métodos

Operacao
+ operacao : String
+ somar(a : double, b : double) : double + somar(a : double, b : double, c : double) : double

```
public class Operacao {

    public String operacao;

    double somar(double a, double b){
        return a + b;
    }

    double somar(double a, double b, double c){
        return a + b + c;
    }

}
```

Classes na Prática

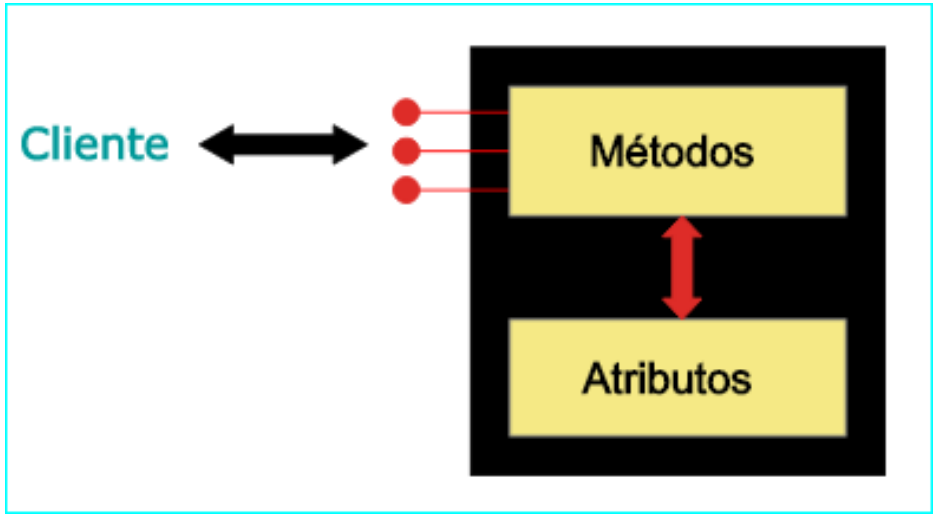
- Sobrecarga de Construtores

```
public class Operacao {

    public String operacao;

    public Operacao(){}

    public Operacao(String operacao){
        this.operacao = operacao;
    }

}
```

Classes na Prática

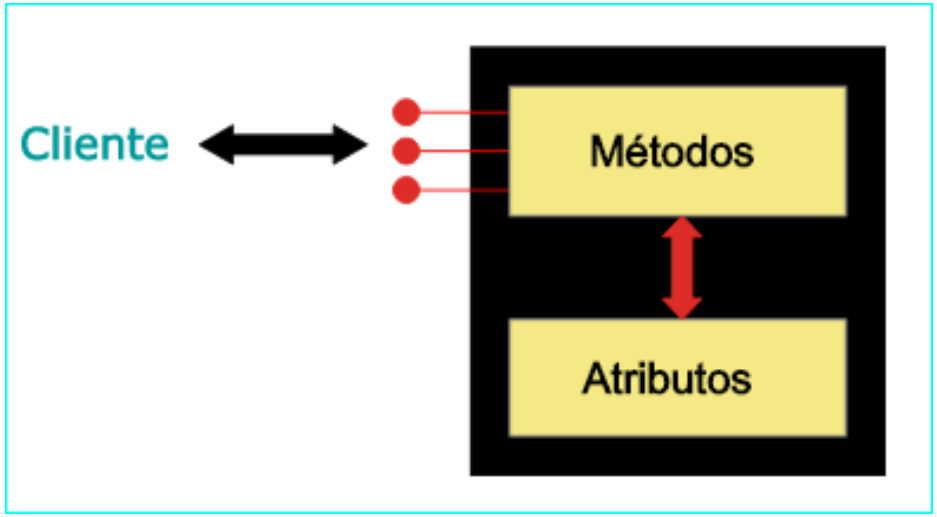
Visões de um objeto

- **Interna**
  - Atributos e métodos da classe que o define
- **Externa**
  - Serviços que o objeto proporciona e como ele interage com outros objetos



Visões de um objeto

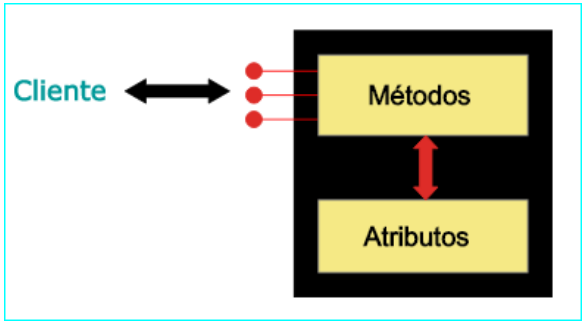
- Externamente, um objeto deve ser visto como uma entidade encapsulada
  - Deve disponibilizar seus serviços sem expor como estes serviços são implementados



Classes na Prática

Encapsulamento

- Proteger dados referentes a um objeto de acesso externo indevido
- Garantir que detalhes internos de implementação de uma classe permaneçam ocultos aos objetos
- Garantir que a definição das regras de negócio se restrinjam à classe
- É garantido por meio dos modificadores de acesso



Classes na Prática

Modificadores de Acesso

- Determinam a visibilidade da classe e de seus membros (atributos e métodos).
- Ao todo são quatro modificadores principais
  - Public
  - Protected
  - Private
  - *Default/Package – (não é atribuído modificador)*

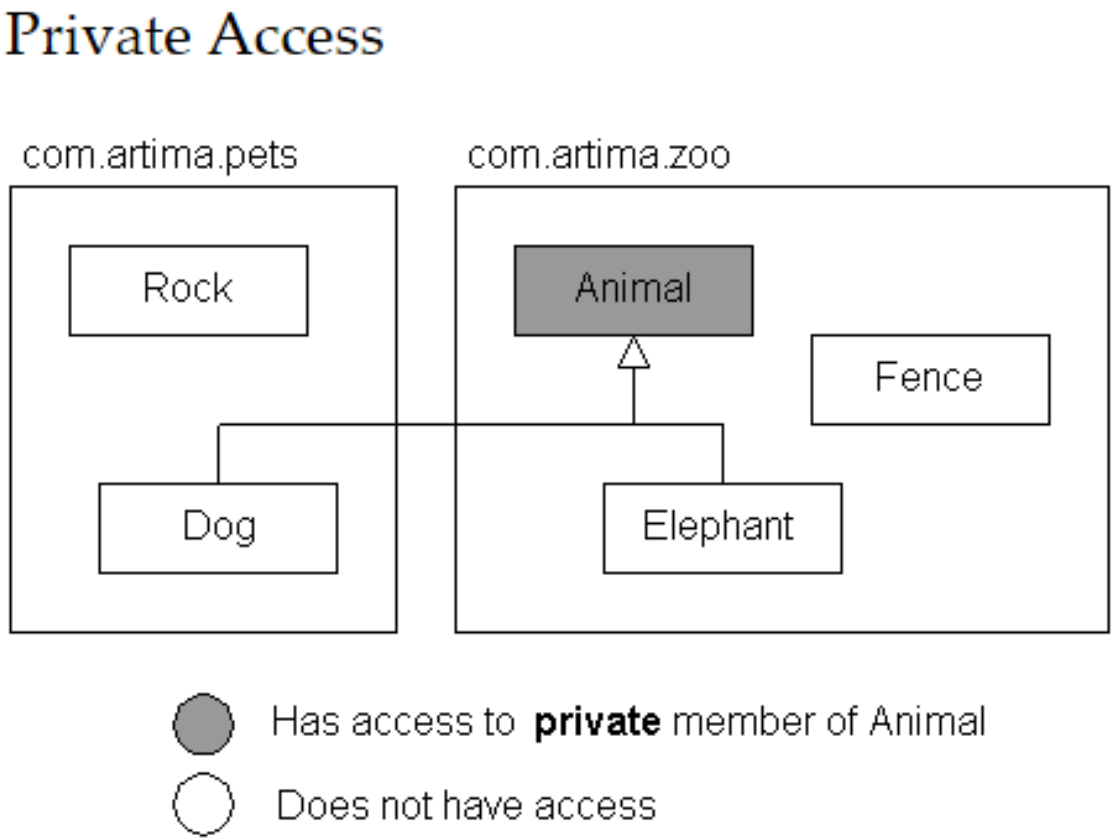
Classes na Prática

Modificadores de Acesso

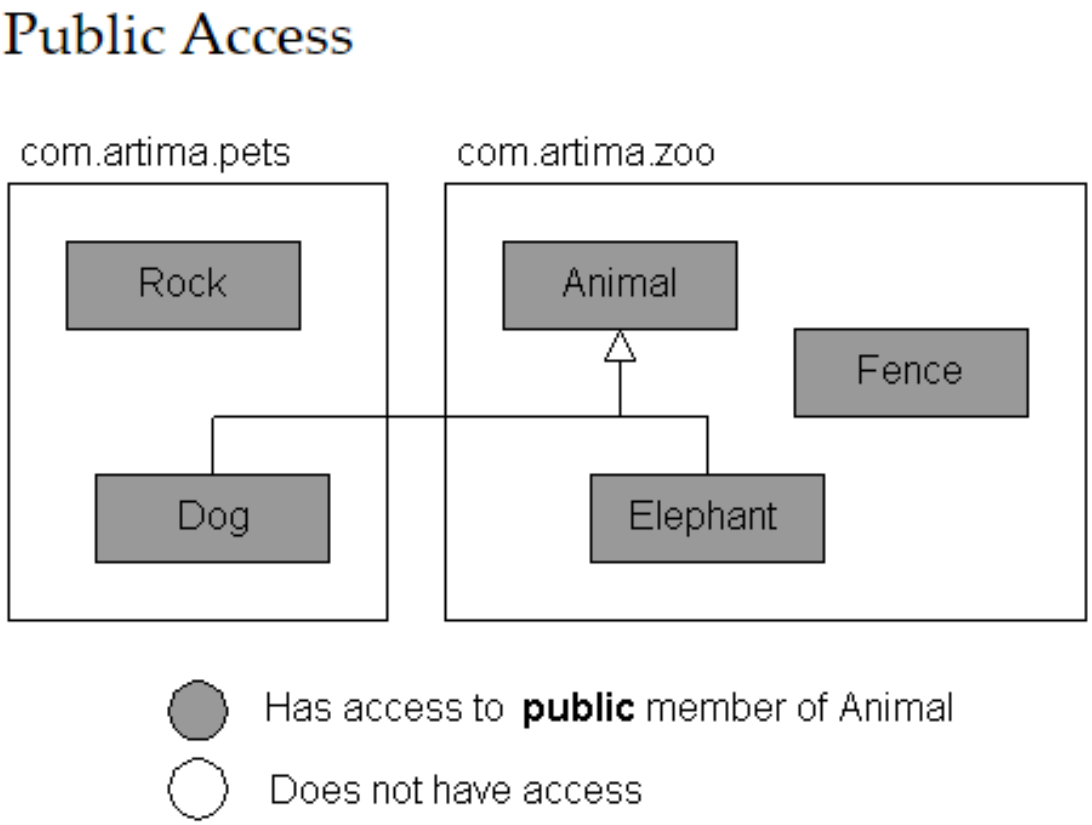
```
[modificador] class Pessoa {
  [modificador] String nome;
  [modificador] int idade;

  [modificador] void imprimirNomeIdade() {
    System.out.println("nome="+nome);
    System.out.println("idade="+idade);
  }
}
```

	private	default	protected	public
mesma classe	sim	sim	sim	sim
mesmo pacote	não	sim	sim	sim
pacotes diferentes (subclasses)	não	não	sim	sim
pacotes diferentes (sem subclasses)	não	não	não	sim

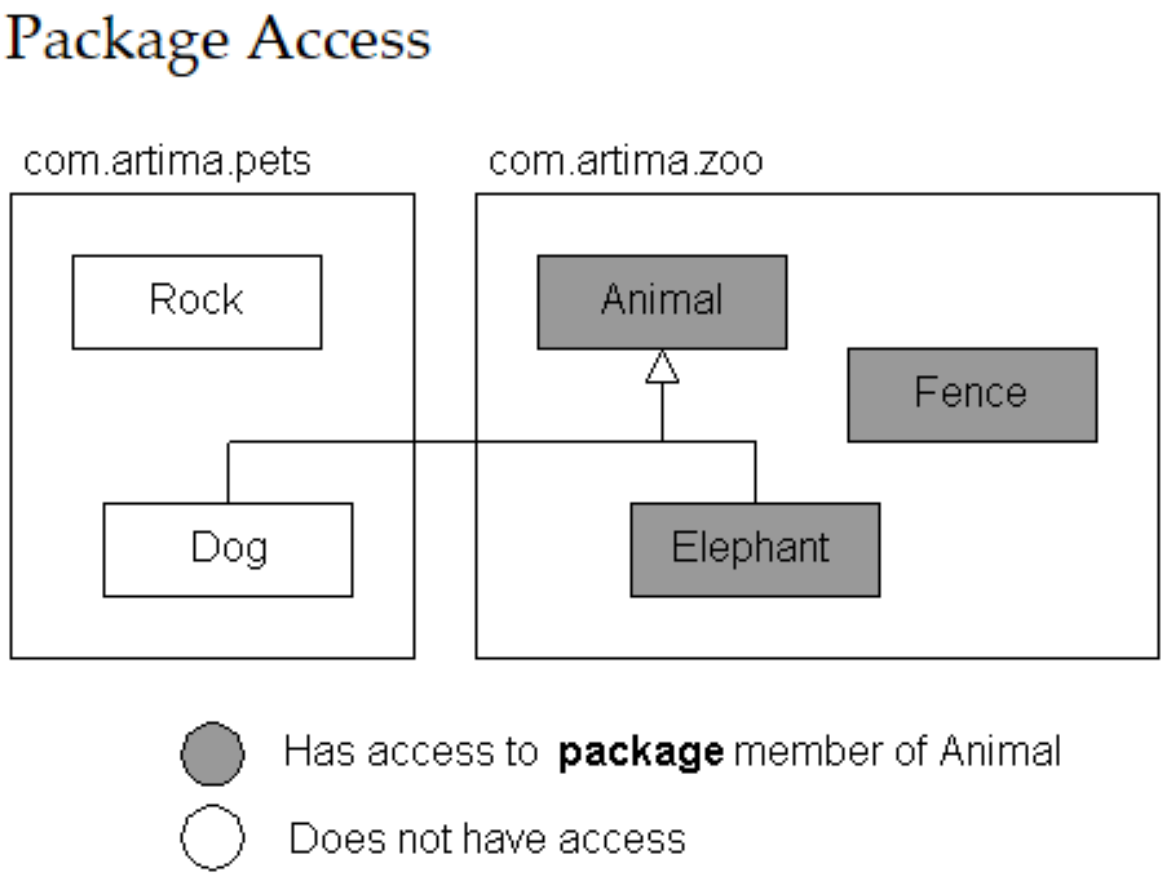
- **Modificadores de Acesso**



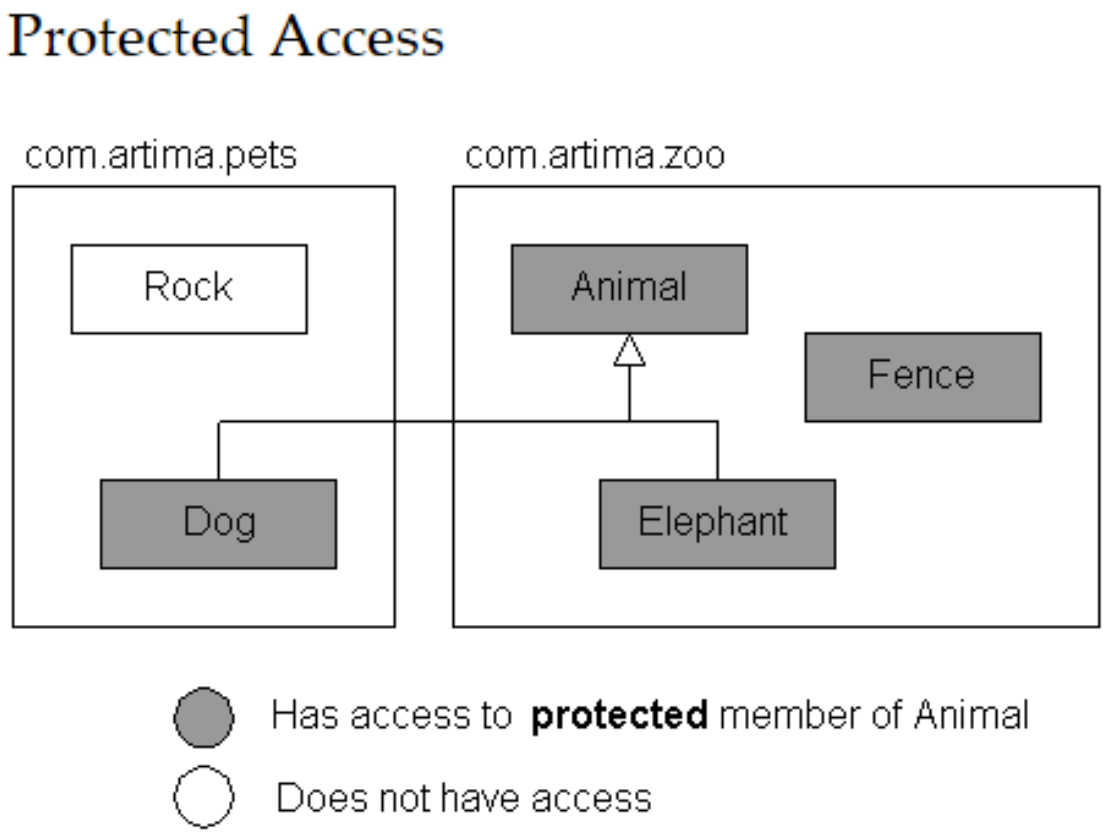
- **Modificadores de Acesso**



Modificadores de Acesso



- **Modificadores de Acesso**



Classes na Prática

- **Encapsulamento**
  - Ocultar detalhes internos da classe
  - Segundo o encapsulamento, os atributos não podem ser acessados/alterados diretamente.
    - Acesso => Atributos ***PRIVATE***
    - Alteração => Métodos ***SET*** e ***GET***.

Classes na Prática

- **Encapsulamento**
  - Passos para criar
    - Modificar o acesso aos atributos para private
    - Criar métodos getter e *setter* apenas se houver necessidade.
      - **Não gere getters e setters indiscriminadamente**
      - **Só gere esses métodos se houver uma necessidade real para a existência deles.**

Classes na Prática

Encapsulamento

- Métodos *get*
  - **Responsável por retornar o valor de uma variável**

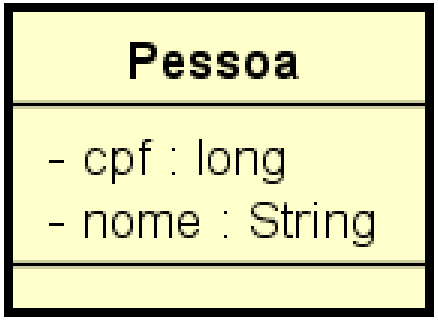
```
tipoAtributo getAtributo(){
    return this.atributo;
}
```

- Métodos *set*
  - **Responsável por atribuir o valor a uma variável**

```
void setAtributo(tipoAtributo atributo) {
        this.atributo = atributo;
}
```

Classes na Prática

- **Encapsulamento**
  - **Exemplo**



Classes na Prática

- **Encapsulamento**
  - **Tornar atributos privados**

```
public class Pessoa {

    private long cpf;
    private String nome;

}
```

Classes na Prática

- **Encapsulamento**
  - **Não será mais possível acessar os atributos diretamente.**

!!ERRO!!

```
public class ExecPessoa {
    public static void main(String args[]){

        Pessoa p = new Pessoa();
        p.cpf = 12345678;
        p.nome = "Fulano";

        System.out.println("Cpf " + p.cpf);
        System.out.println("Nome: " + p.nome);

    }
}
```

cpf has private access in Pessoa

(Alt-Enter mostra dicas)

Classes na Prática

Encapsulamento

- Gerar ***getter*** e ***setter*** apenas para os atributos necessários

```
public class Pessoa {

    private long cpf;
    private String nome;

    public long getCpf() {
        return cpf;
    }
    public void setCpf(long cpf) {
        this.cpf = cpf;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

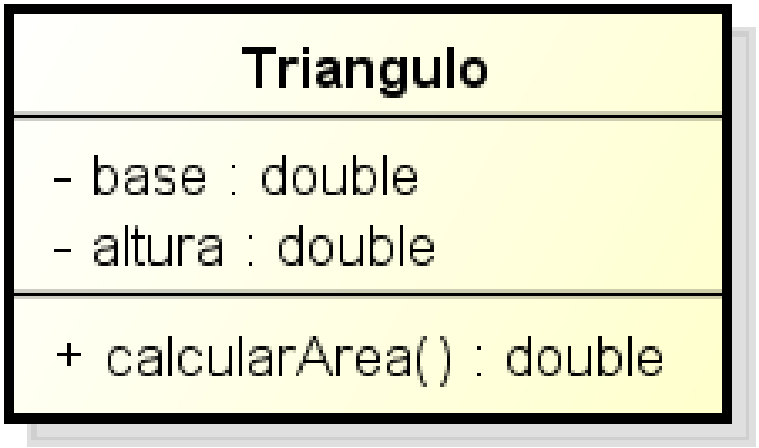
Classes na Prática

- **Encapsulamento**
  - **Acesso correto**

```
public class ExecPessoa {
    public static void main(String args[]){

        Pessoa p = new Pessoa();
        p.setCpf(12345678);
        p.setNome("Joyce");

        System.out.println("Cpf " + p.getCpf());
        System.out.println("Nome: " + p.getNome());

    }
}
```

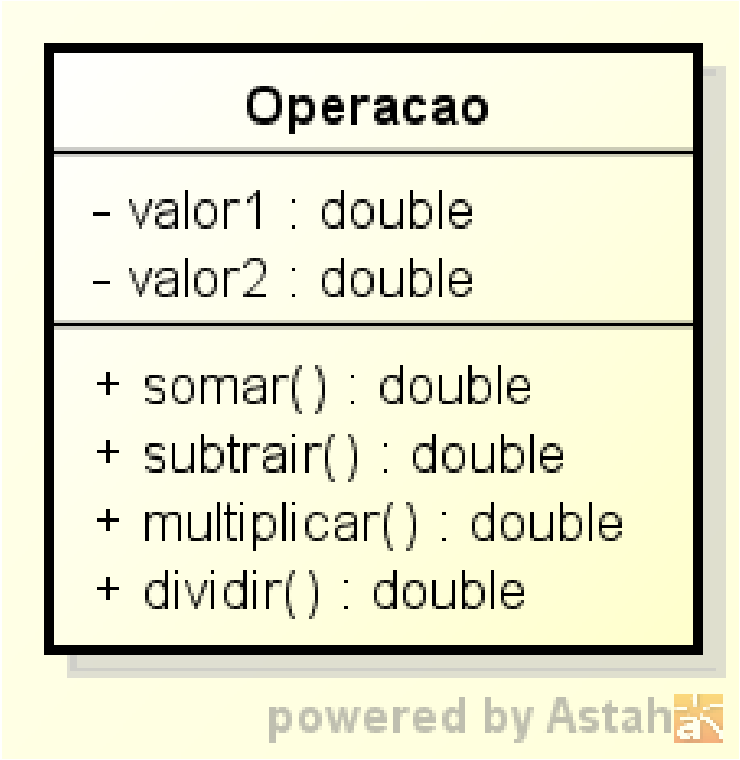
Exercício 01

- Encapsulamento + Construtor



Exercício 02

- Encapsulamento + Construtor

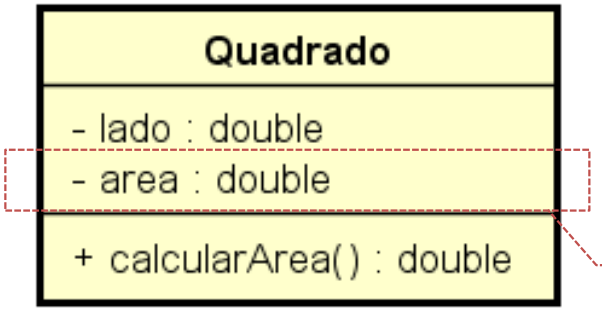


Classes na Prática

Encapsulamento

Observações

- Atributos com regra de negócio associada **não** devem possuir métodos *setters*.
- Gerar *setters* indevidos:
  - Quebra o encapsulamento
  - Vai permitir que outras classes definam regras de negócio a partir dos *setters*
  - A mesma regra de negócio pode ficar espalhada em diversas classes do projeto



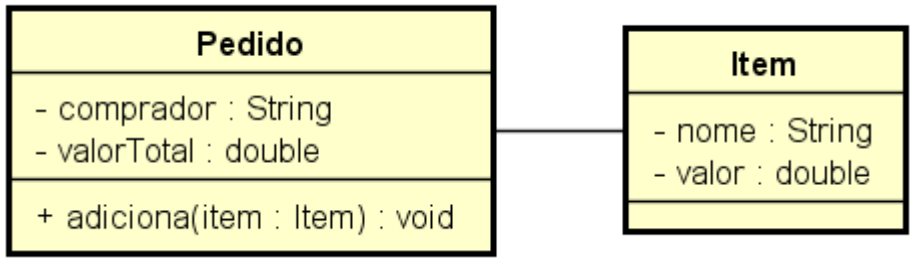
Não deve ter *setter*

q.area = lado*lado
quadrado.setArea(lado*lado)

- As classes devem ser auto-contidas
  - Regras de negócio devem se restringir a própria classe

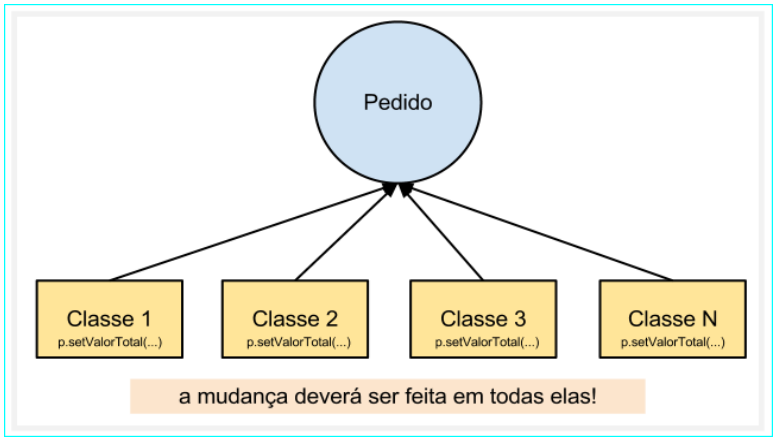
Classes na Prática

Exemplo de encapsulamento mal feito




```
Pedido p = new Pedido("Cliente Fulano");
Item novoItem = new Item("Arroz", 2.50);
//atualiza valor do pedido
p.setValor(p.getValor() + novoItem.getValor());
```

Regra de negócio definida durante a atribuição de valores



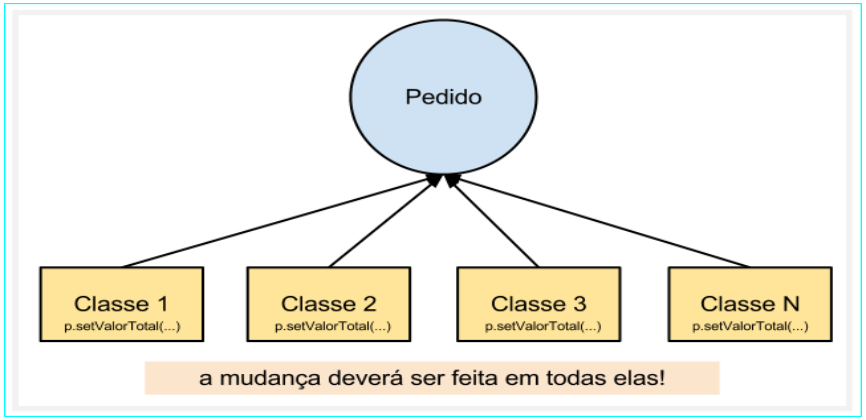
Regra de negócio espalhada pelo projeto

Classes na Prática

Exemplo de encapsulamento mal feito

- Mudanças nas regras de negócios causam alteração por todo o projeto dificultando a manutenção

```
Pedido p = new Pedido("Joyce", 0.0);
Item novoItem = new Item("Arroz", 2.50);
//atualiza valor do pedido
if (novoItem.getValor() > 1000){
    p.setValorTotal(p.getValorTotal() + novoItem.getValor() * 0.95);
}else{
    p.setValorTotal(p.getValorTotal() + novoItem.getValor());
}
```

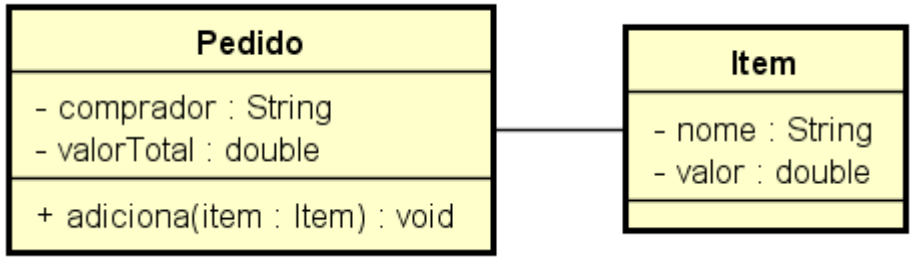


Classes na Prática

Deve ser responsabilidade exclusiva da classe encapsular regras de negócio

Encapsulamento – Solução Ideal

- Criar métodos mais claros para a atualização dos valores




```
class Pedido {
  private String comprador;
  private double valorTotal;
  // outros atributos

  public String getComprador() { return comprador; }
  public double getValorTotal() { return valorTotal; }

  public void adiciona(Item item) {
    if(item.getValor() < 1000) this.valorTotal += item.getValor();
    else this.valorTotal += item.getValor() * 0.95;
  }
}
```

Classes na Prática

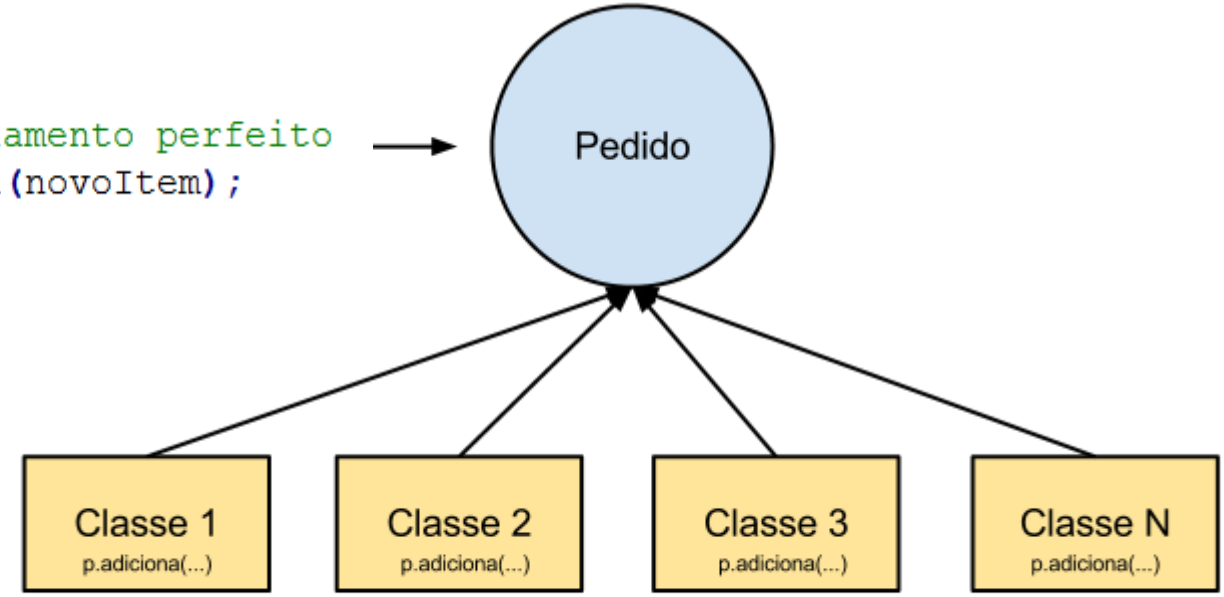
- **Encapsulamento – Solução Ideal**
  - Quando sabemos **O QUÊ** um método faz mas não sabemos exatamente **COMO** ele faz, dizemos que esse comportamento está encapsulado!

```
//classe de execucao..

//solucao antiga
if (novoItem.getValor() > 1000){
    p.setValorTotal(p.getValorTotal() + novoItem.getValor() * 0.95);
}else{
    p.setValorTotal(p.getValorTotal() + novoItem.getValor());
}

//solucao nova
//encapsulamento perfeito
p.adiciona(novoItem);
```

Classes na Prática

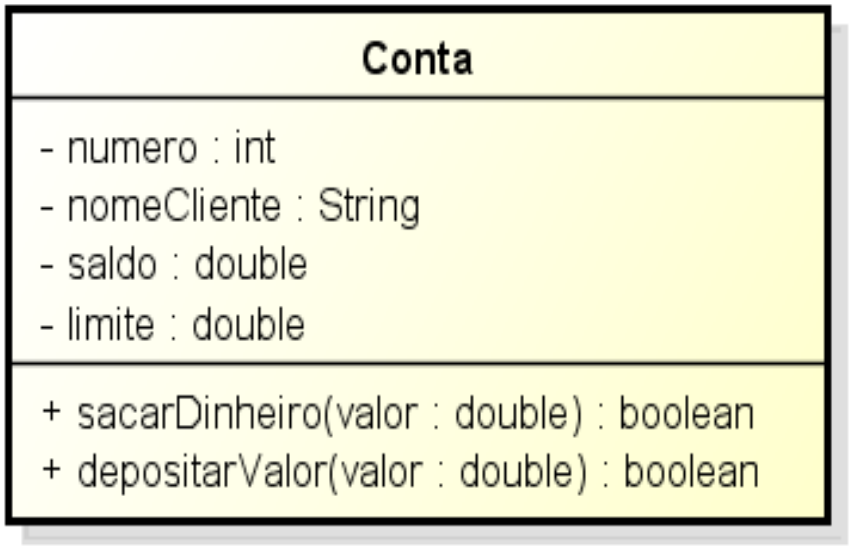
- **Encapsulamento – Solução Ideal**
  - A partir do momento que outras classes não sabem como a classe principal faz o seu trabalho, significa que as mudanças ocorrerão apenas em um lugar!

```
//encapsulamento perfeito
p.adiciona(novoItem);
```



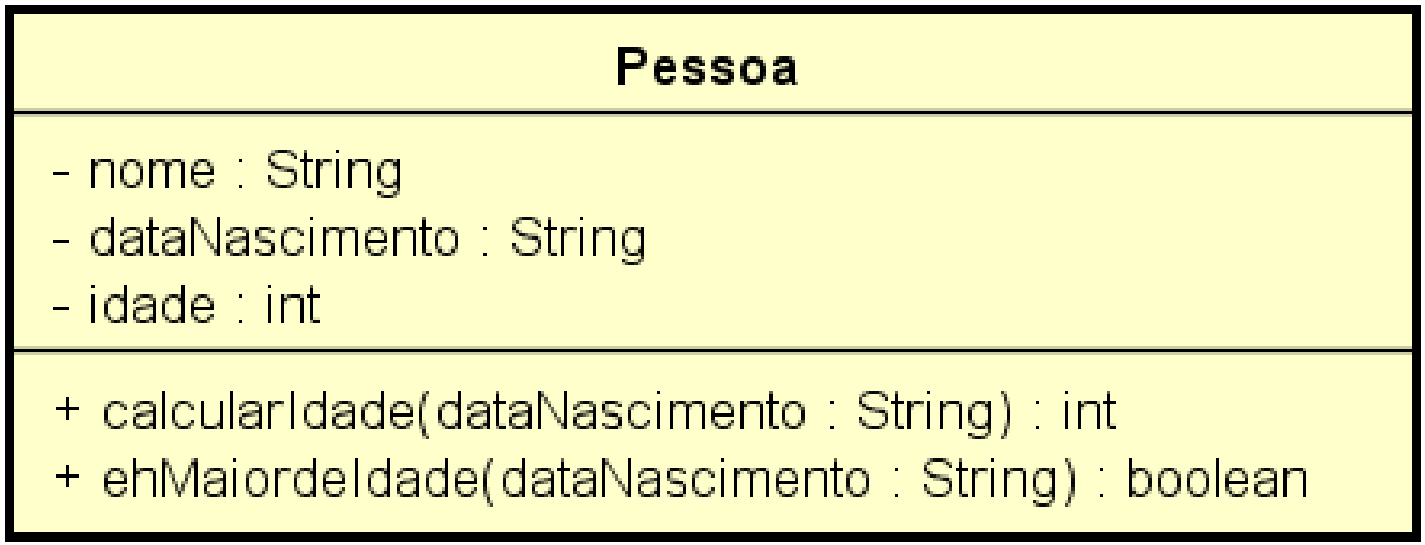
Exercício 03

- Encapsulamento + Construtor



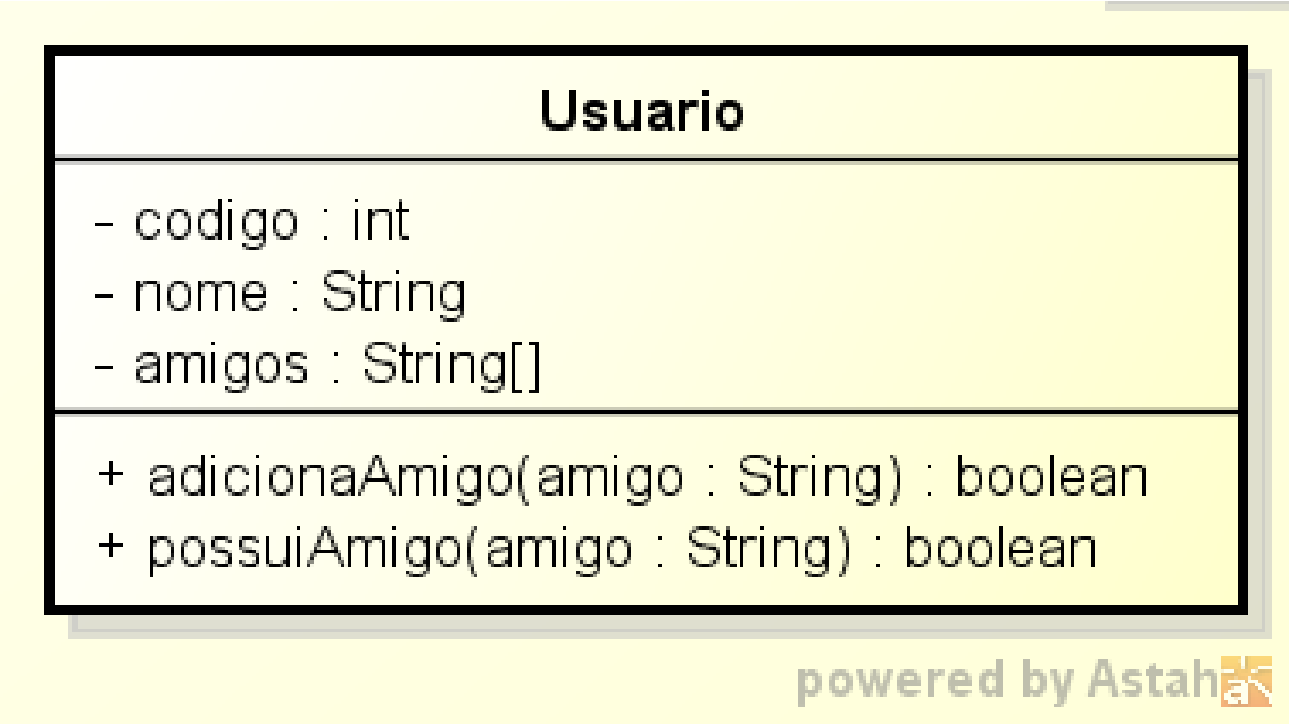
Exercício 04

- Encapsulamento + Construtor



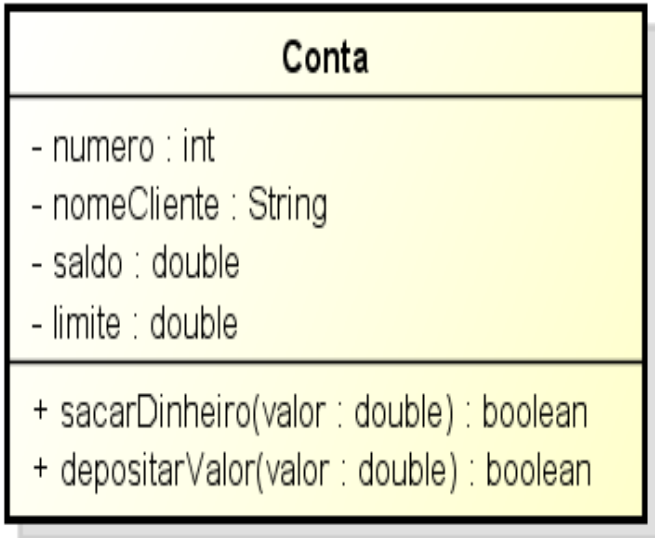
Exercício 05

- Encapsulamento + Construtor



Array de Referências

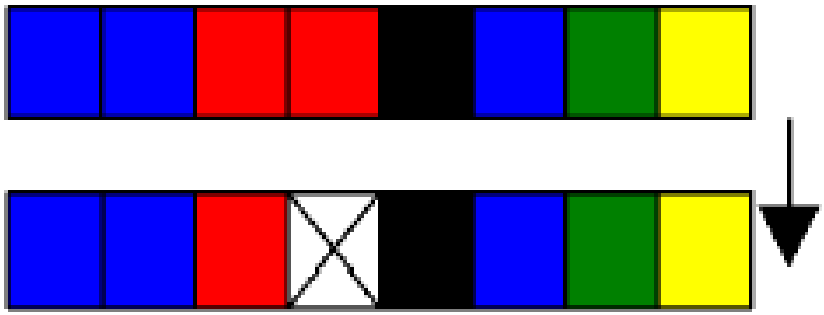
```
Scanner s = new Scanner(System.in);

String[] nomes = new String[5];

for (int i = 0; i < nomes.length; i++) {
    nomes[i] = s.nextLine();
}

for (String nome : nomes) {
    System.out.println(nome);
}
```



- Crie o código para ler, percorrer e exibir um array de objetos do tipo Conta

Limitações do Array

- Não podemos redimensionar um array;
- Complexidade para saber quantos elementos foram inseridos?
- Complexidade para saber quais posições estão livres?

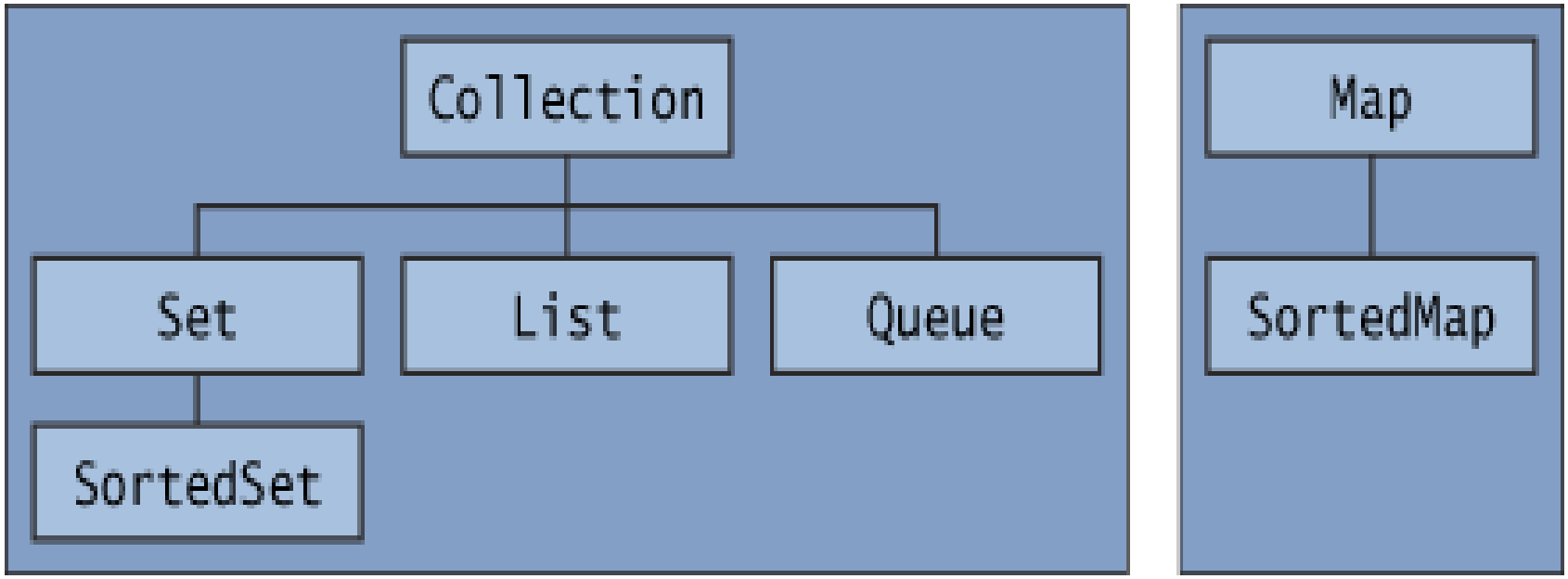


Retire a quarta Conta

```
conta[3] = null;
```

Collections API

- **Conjunto de implementações que representam estruturas de dados avançadas.**



Collections API

- Listas:
  - import java.util.List;
  - import java.util.ArrayList;
    - Criando

```
List<Integer> lista = new ArrayList();
```

    - Adicionando

```
lista.add(100);
lista.add(1000);
lista.add(10000);
```

    - Retornando o tamanho da lista

```
lista.size();
```

Collections API

- Listas:
  - import java.util.List;
  - import java.util.ArrayList;
    - Verificando se valor existe

```
boolean valorExiste = lista.contains(101);
```

    - Recuperando o valor de uma posição

```
lista.get(0);
```

    - Atualizando valor de uma posição

```
lista.set(0,101);
```

Collections API

- Listas:
  - import java.util.List;
  - import java.util.ArrayList;
    - Percorrendo lista

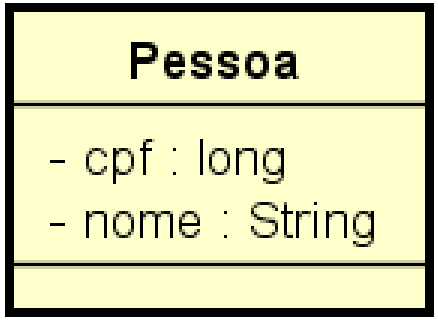
```
//opcao I
for (int i = 0; i < lista.size(); i++) {
    System.out.println(lista.get(i));
}
//opcao II
for (Integer item : lista) {
    System.out.println(item);
}
//opcao III
lista.forEach((item) -> {
    System.out.println(item);
});
```

Exercício 06

- Criar uma lista de números
- Adicionar números à lista
- Mostrar quantidade de números presentes na lista
- Listar todos os números adicionados
- Exibir a soma total dos números da lista

Collections API

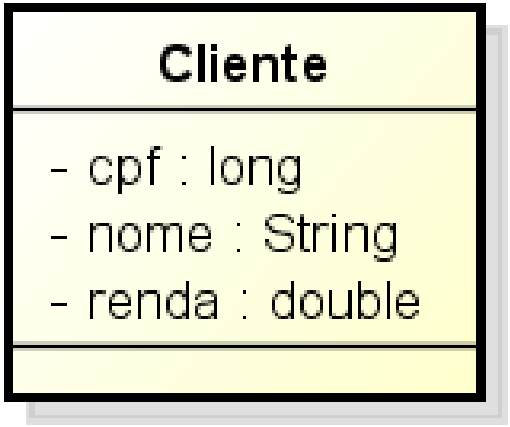
- Listas de Referências




```
List<Pessoa> listaPessoas = new ArrayList();
listaPessoas.add(new Pessoa(12345678, "Fulano"));
listaPessoas.add(new Pessoa(87654321, "Ciclano"));
```

Exercício 07

- Criar uma lista de clientes
- Adicionar clientes à lista
- Mostrar a quantidade de clientes presentes na lista
- Listar todos os clientes presentes na lista
- Exibir a renda média dos clientes presentes na lista



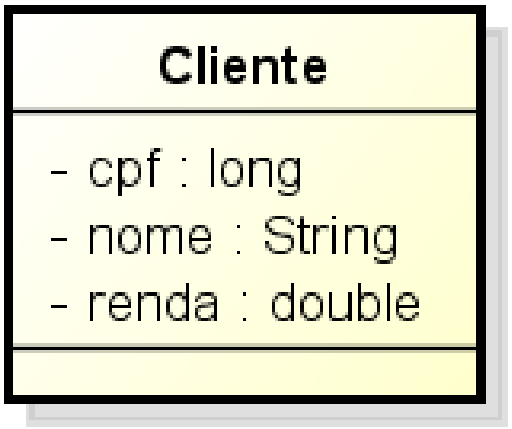
Collections API

- Buscando um valor na lista
  - Método *contains()*

```
List<String> listaNomes = new ArrayList();
listaNomes.add("Fulano");
listaNomes.add("Ciclano");
listaNomes.add("Beltrano");

boolean temCiclano = listaNomes.contains("Ciclano");
```

Collections API

- Buscando um valor na lista
  - Método *contains()*
    - *Como verificar se um cliente já foi adicionando à lista*



Collections API

- Buscando um valor na lista
  - Método *contains()*
    - Incluir no corpo da classe Cliente uma sobrescrita para o método ***equals()***

```
@Override
public boolean equals(Object obj) {
    Cliente c = (Cliente) obj;
    return (c.cpf == this.cpf);

}
```

    - Exemplo de chamada na classe de execução

```
boolean temMaria =
        lista.contains(new Cliente("Maria"));
```

Collections API

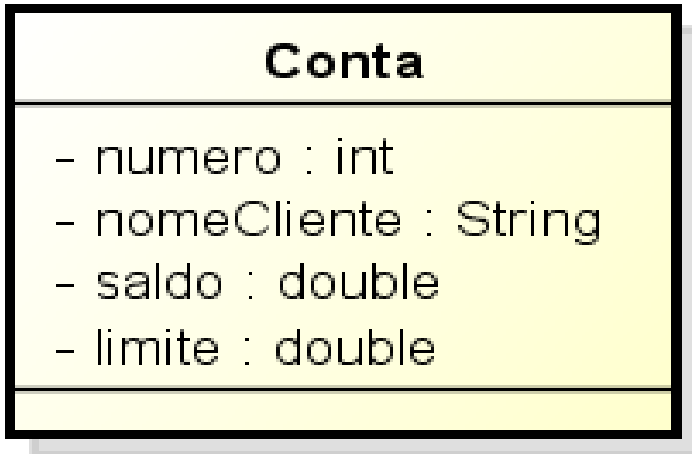
- Ordenando
  - Método *sort()*

```
List<String> listaNomes = new ArrayList();
listaNomes.add("Fulano");
listaNomes.add("Ciclano");
listaNomes.add("Beltrano");

Collections.sort(listaNomes);
```

```
Collections.sort(listaNomes, Collections.reverseOrder());
```

Collections API

- Ordenando
  - Método *sort()*
    - *Como ordenar uma lista de contas?*



Collections API

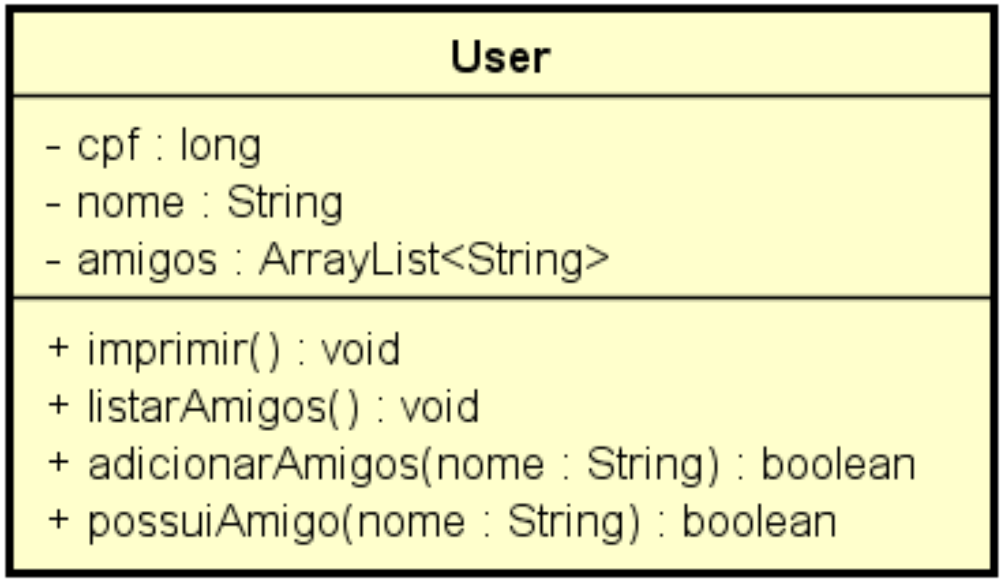
- Ordenando
  - Método *sort()*

```
public class ContaCorrente implements Comparable<ContaCorrente> {
```

```
public int compareTo(ContaCorrente outraConta) {
    if(this.saldo > outraConta.saldo){
      return 1;
    }else if(this.saldo < outraConta.saldo){
      return -1;
    }else{
      return 0;
    }
}
```

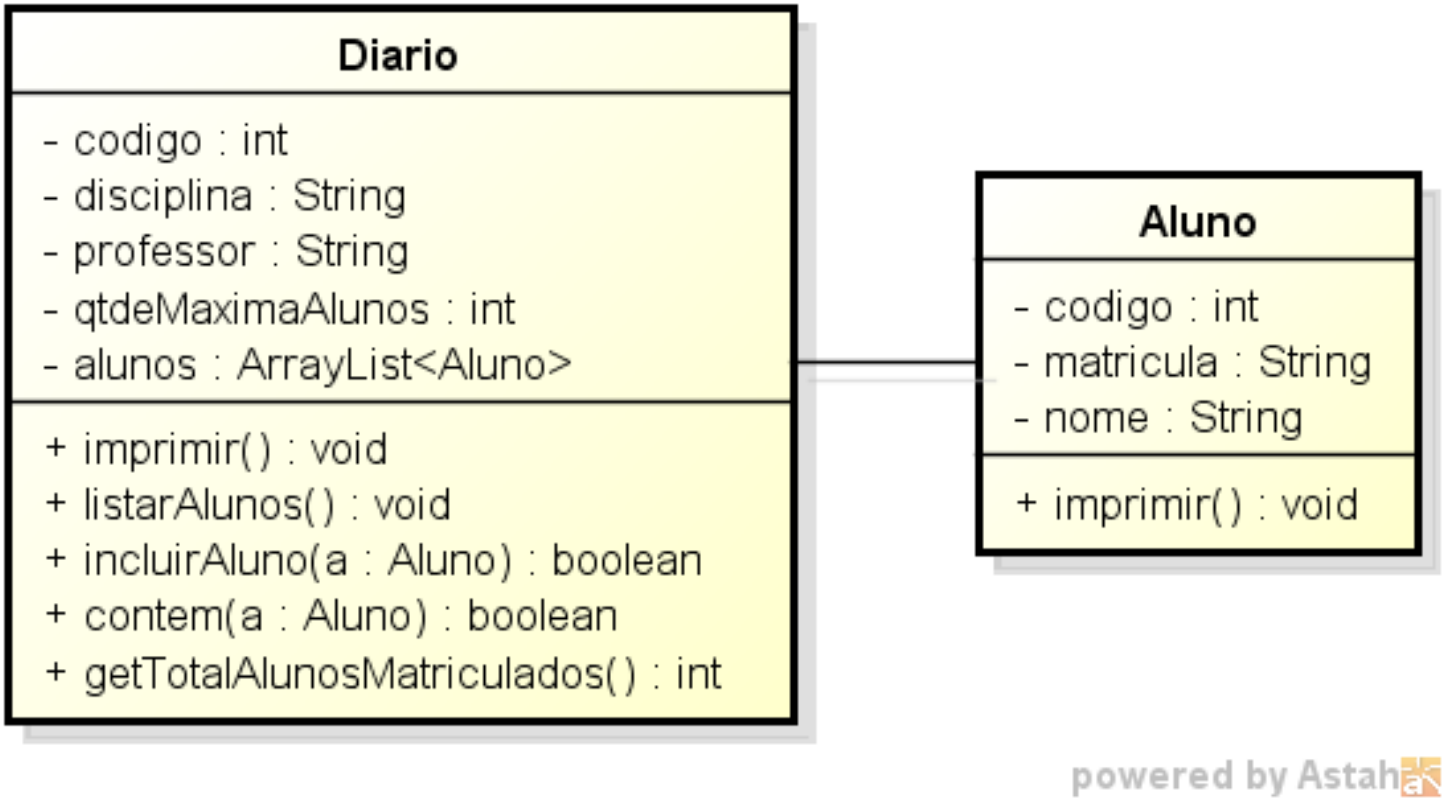
```
Collections.sort(contas);
```

Exercício 08



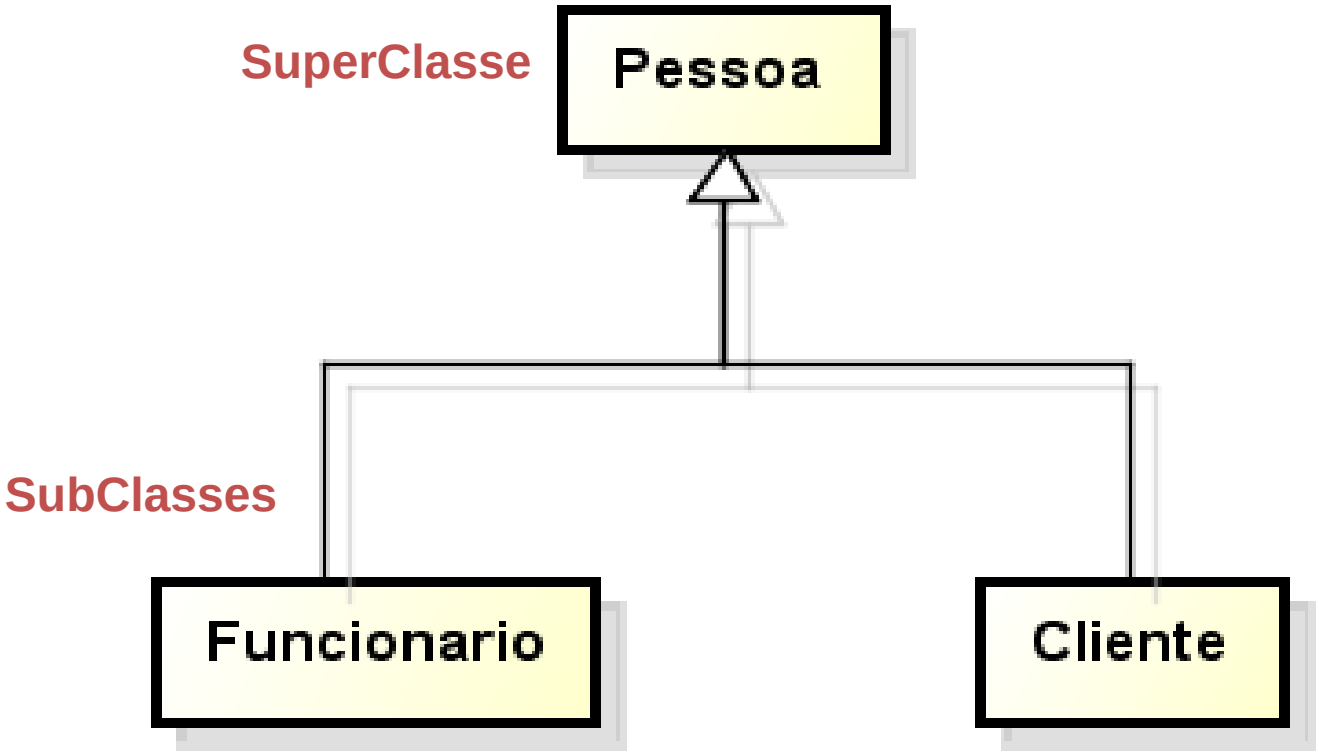
- Crie uma lista de usuários
- Adicionar valores à lista
- Verificar se um usuário foi adicionado à lista
- Ordene a coleção pelo nome do usuário e imprima o resultado

Exercício 09



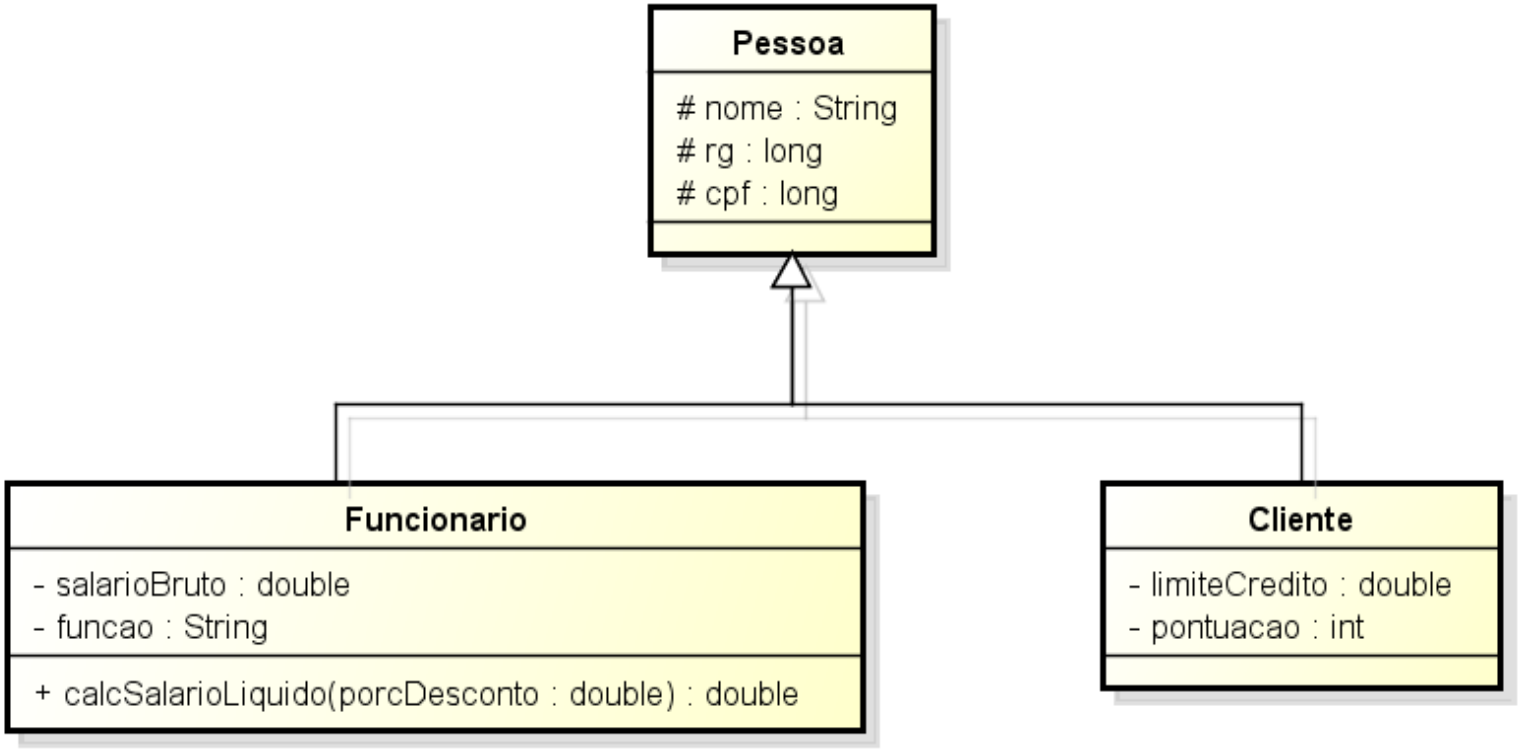
Conceitos - Herança

- Permite criar novas classes a partir de classes já existentes.
- Reflete um relacionamento de especialização



Conceitos - Herança

- Todos os métodos e atributos (***public* e *protected***) são herdados pelas subclasses
- Os construtores não são herdados.



Herança na Prática

- Em Java, a herança é conseguida através da palavra ***extends***;

```
class MyClass {
}
```



```
class MyClass
        extends Object {
}
```

```
class ClasseA {
}
```

```
class ClasseB
        extends ClasseA {
}
```

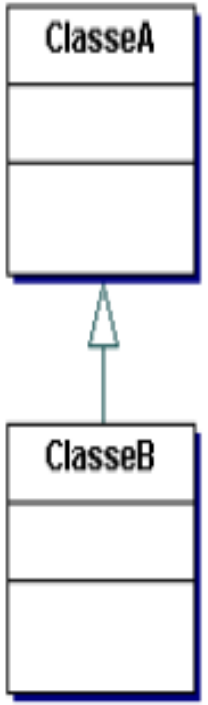
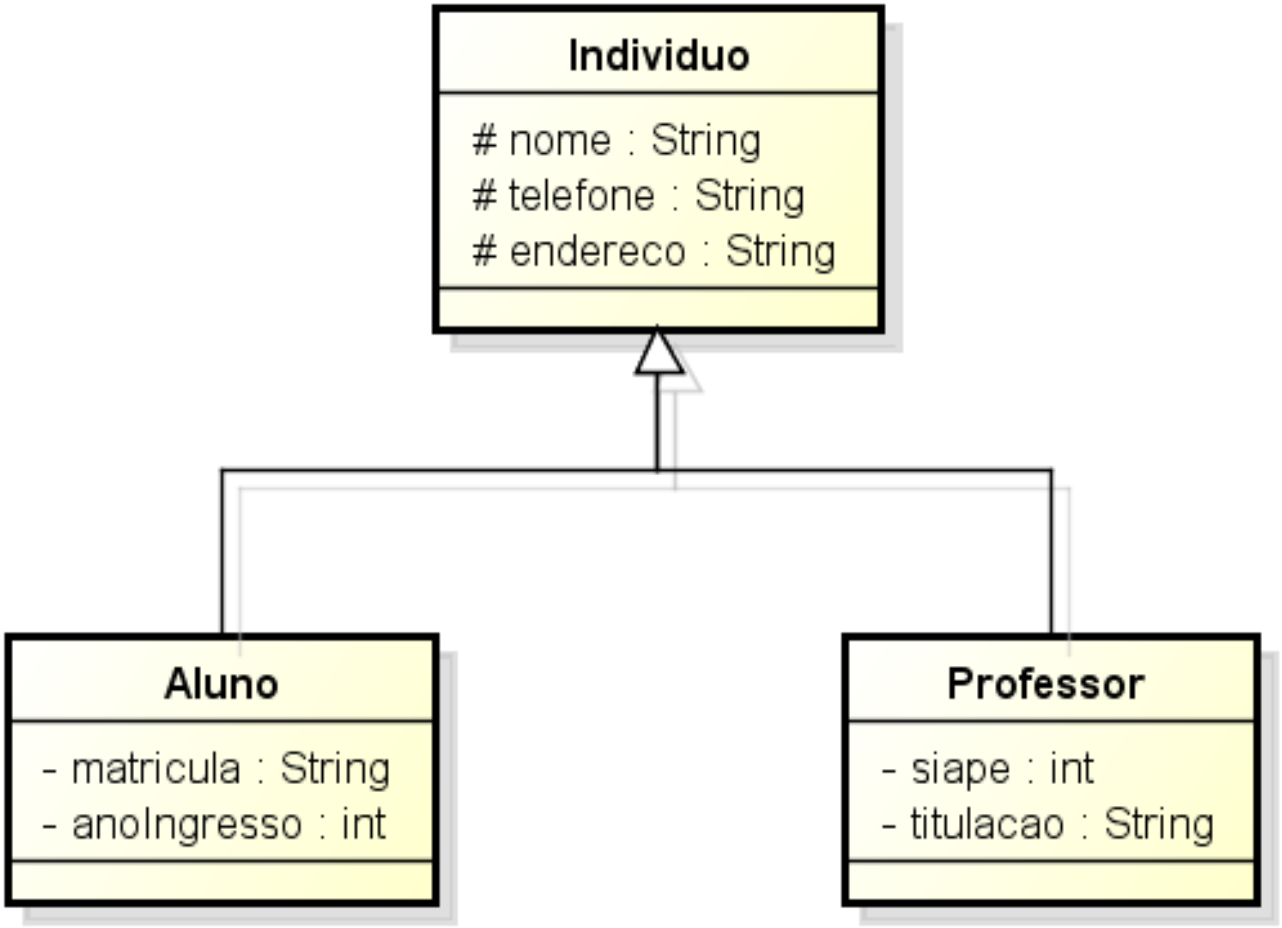


Diagrama UML

Herança na Prática



Herança na Prática

```
 8    public class Individuo {
 9
10        protected String nome;
11        protected String telefone;
12        protected String endereco;
13
14        public String getNome() {...3 lines }
17
18        public void setNome(String nome) {...3 lines }
21
22        public String getTelefone() {...3 lines }
25
26        public void setTelefone(String telefone) {...3 lines }
29
30        public String getEndereco() {...3 lines }
33
34        public void setEndereco(String endereco) {...3 lines }
```

Herança na Prática

```
public class Aluno extends Individuo {

    private String matricula;
    private int anoIngresso;

    public String getMatricula() {...3 lines }

    public void setMatricula(String matricula) {...3 lines }

    public int getAnoIngresso() {...3 lines }

    public void setAnoIngresso(int anoIngresso) {...3 lines }

}
```

Herança na Prática

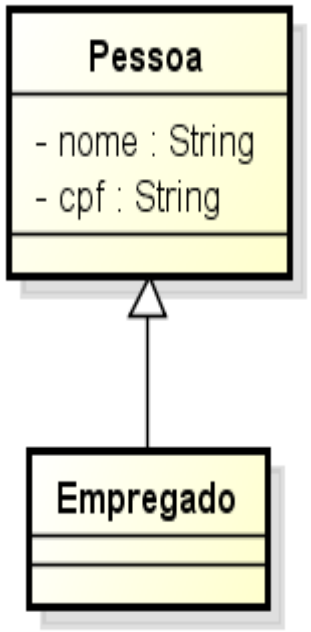
```
public class ExecAluno {

    public static void main(String args[]){

        Aluno aluno = new Aluno();

        aluno.setNome("Fulano");
        aluno.setTelefone("(00)0000-0000");
        aluno.setEndereco("Rua X, Bairro Y");
        aluno.setMatricula("123456789");
        aluno.setAnoIngresso(2016);

        System.out.println("Nome: " + aluno.getNome() +
                " Matrícula: " + aluno.getMatricula());

    }
}
```

Herança na Prática

- O construtor da superclasse é chamado automaticamente, se outra chamada não for feita.
- A palavra ***super*** *referencia a superclasse.*

```
public class Pessoa {
  protected String nome;
  protected String cpf;

  public Pessoa() {}
}
```

```
public class Empregado extends Pessoa {
  public Empregado() {
    super(); //ocorre automaticamente
  }
}
```



Herança na Prática

- Modificando construtor

```
public class Individuo {

    protected String nome;
    protected String telefone;
    protected String endereco;

    public Individuo(String nome, String telefone, String endereco) {
        this.nome = nome;
        this.telefone = telefone;
        this.endereco = endereco;
    }

    public String getNome() {...3 lines }
```

Herança na Prática

- Modificando construtor

```
constructor Individuo in class Individuo cannot be applied to given types;
  required: String,String,String
  found: no arguments
  reason: actual and formal argument lists differ in length
----
(Alt-Enter shows hints)
```

```
public class Aluno extends Individuo {

    private String matricula;
    private int anoIngresso;

    public String getMatricula() {
        return matricula;
    }
```

Herança na Prática

- Modificando construtor

```
public class Aluno extends Individuo {

    private String matricula;
    private int anoIngresso;

    public Aluno(String matricula, int anoIngresso,
                 String nome, String telefone, String endereco) {
        super(nome, telefone, endereco);
        this.matricula = matricula;
        this.anoIngresso = anoIngresso;
    }

    public String getMatricula() {
        return matricula;
    }
```

Herança na Prática

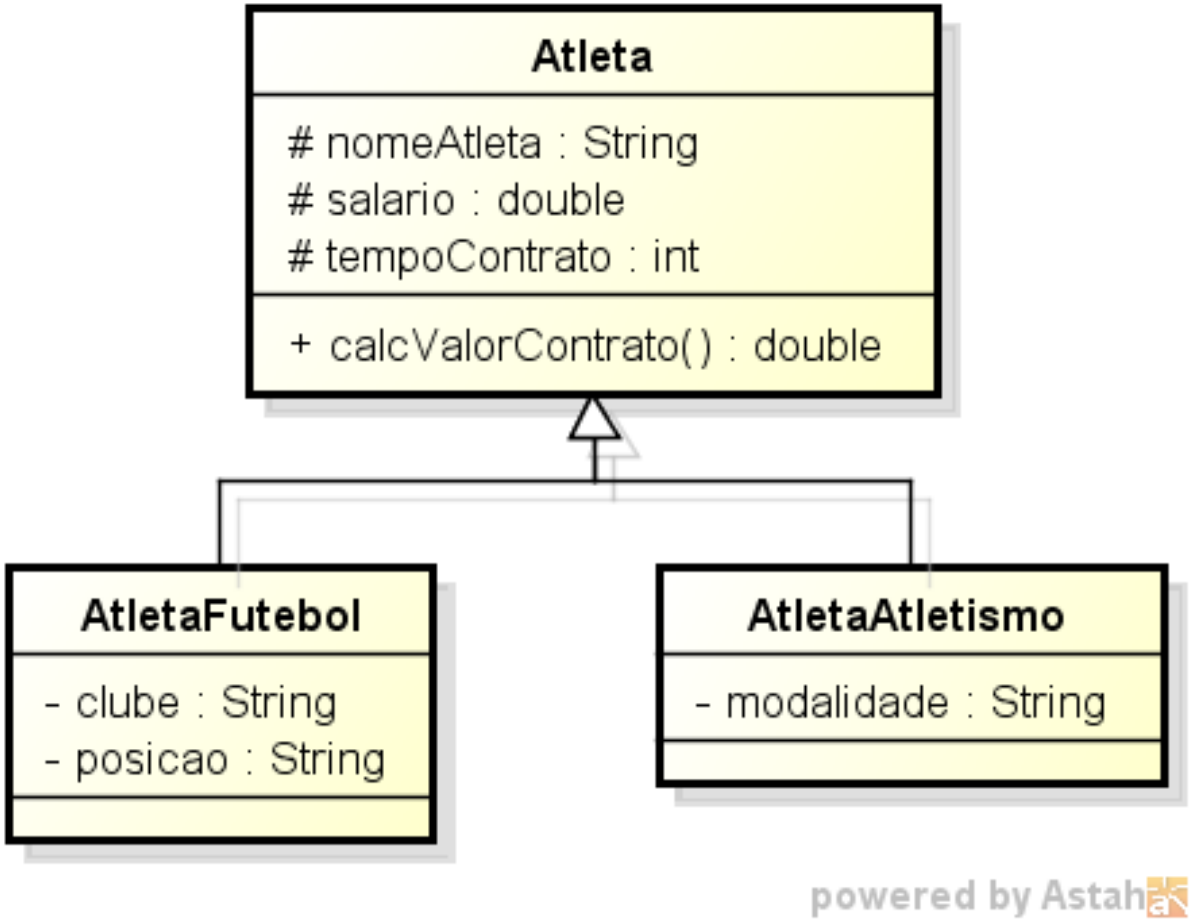
- Modificando construtor

```
public class ExecAluno {

    public static void main(String args[]){

      Aluno al = new Aluno("987654321", 2016,
                           "Beltrano", "0000-0000", "Rua Z");

      System.out.println("Nome: " + al.getNome() +
              " Matrícula: " + al.getMatricula());

    }
}
```

Exercício 10

- HERANÇA

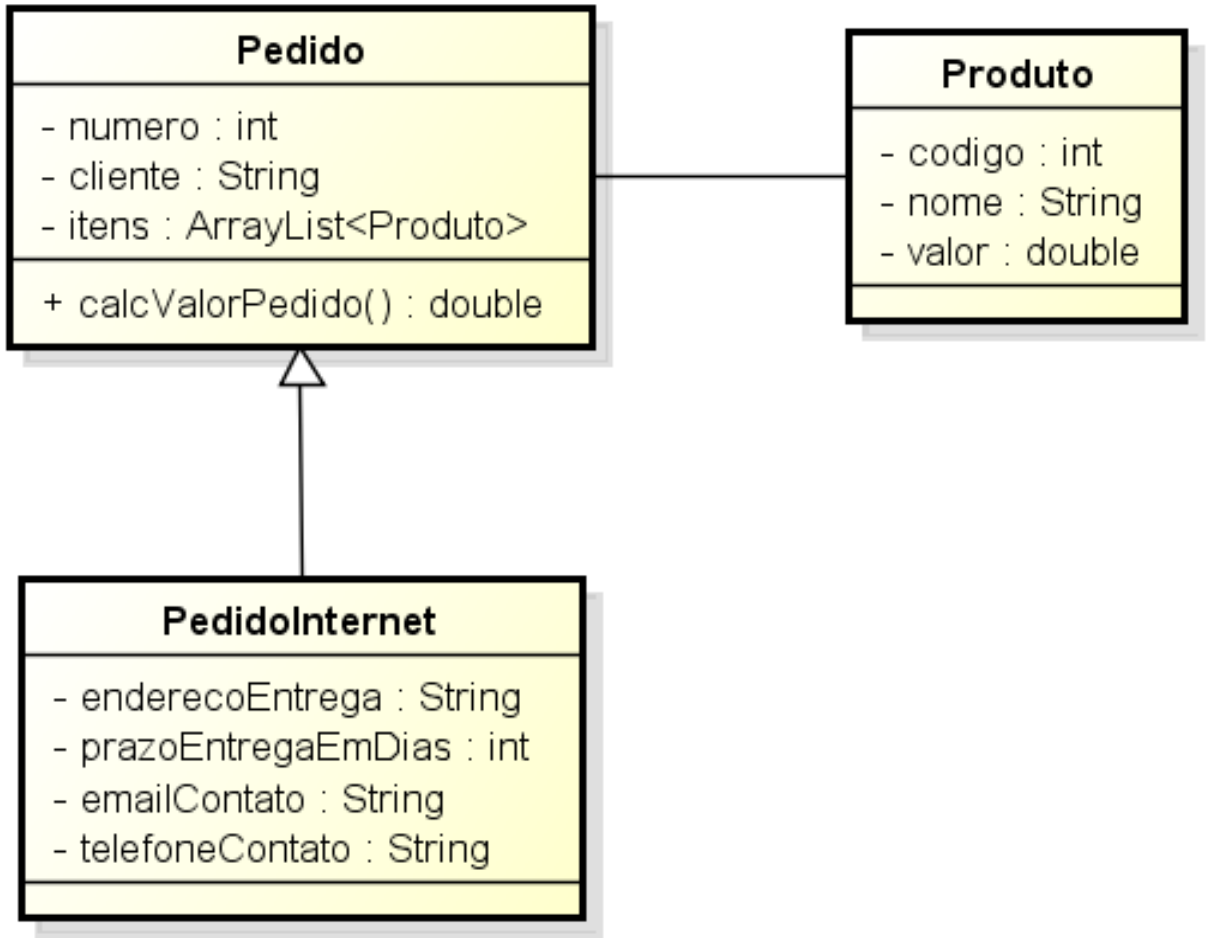
calcValorContrato() =
*salario * tempoContrato*



Exercício 11

- HERANÇA

calcValorPedido() = Soma dos valores de todos os itens do pedido

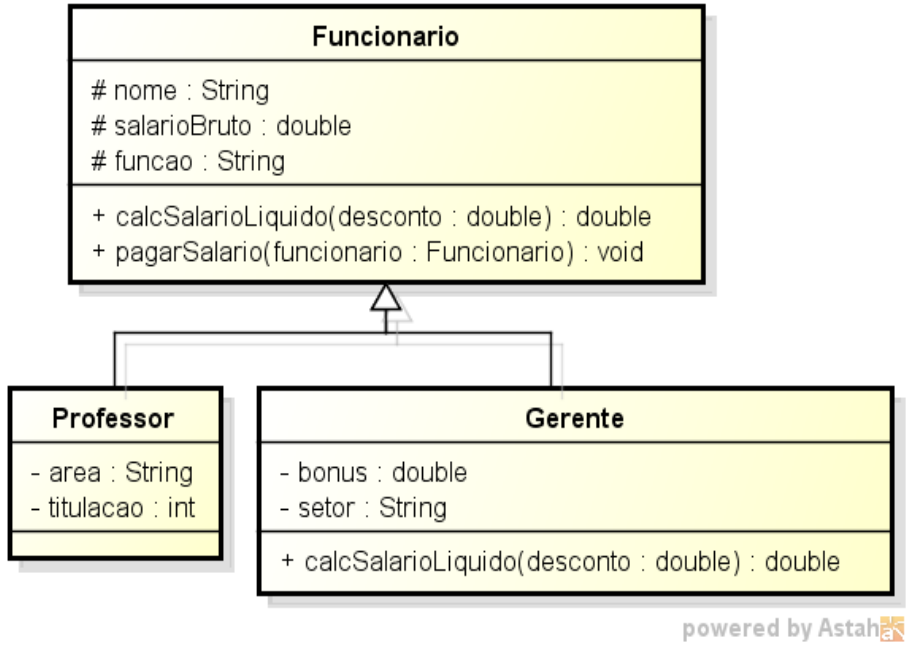


Polimorfismo Dinâmico

- Definições
  - Capacidade de um objeto poder ser referenciado de várias formas.

```
Funcionario f = new Funcionario();
Funcionario p = new Professor();
Funcionario g = new Gerente();
```

O que guarda a variável do tipo Funcionário?



Polimorfismo Dinâmico

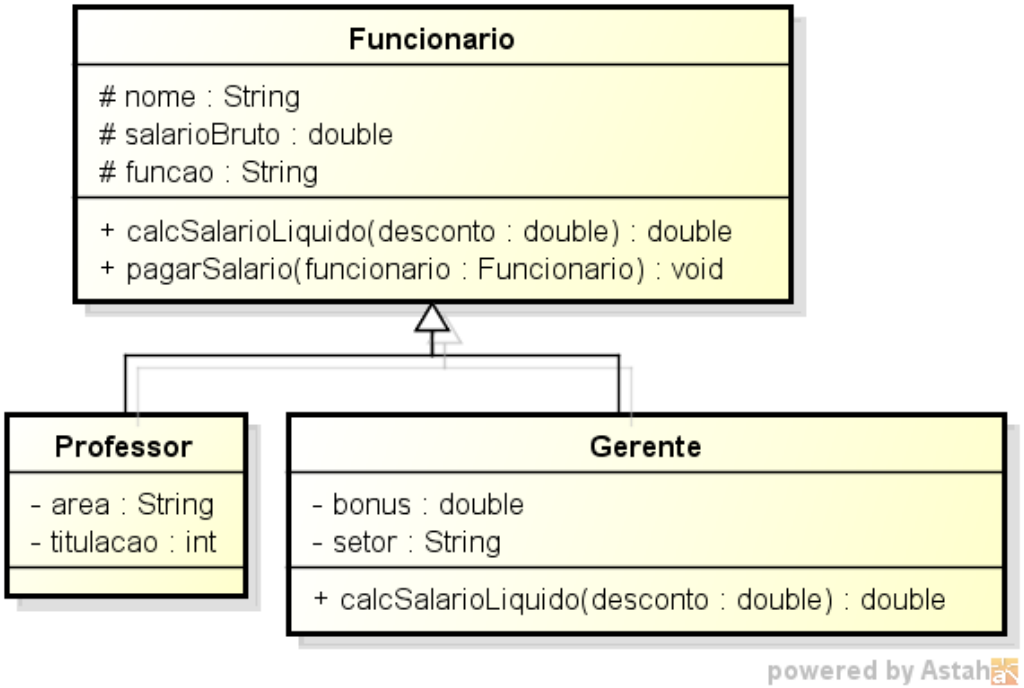
- Definições
  - Denota uma situação na qual objetos de um mesmo tipo podem se comportar de formas distintas, dependendo do seu tipo de criação.

```
Funcionario f = new Funcionario();
Funcionario p = new Professor();
Funcionario g = new Gerente();

f.calcSalarioLiquido();
p.calcSalarioLiquido();
g.calcSalarioLiquido();
```

  - Princípio a partir do qual objetos derivados de uma mesma classe são capazes de invocar métodos que, **embora apresentem a mesma assinatura**, se comportam de maneira diferente.

Polimorfismo Dinâmico

- Se dá pela redefinição/sobrescrita (mesma assinatura) de métodos herdados



- O comportamento executado pelo método será definido em tempo de execução...

Polimorfismo Dinâmico

- Qual a utilidade disso?

```
boolean pagarFuncionario(Funcionario f){
    double salario = f.calcSalarioLiquido();
    return depositar(salario);
}
```

```
Professor professor = new Professor();
Gerente gerente = new Gerente();
```

```
pagarFuncionario(gerente);
pagarFuncionario(professor);
```

- ***não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele.***

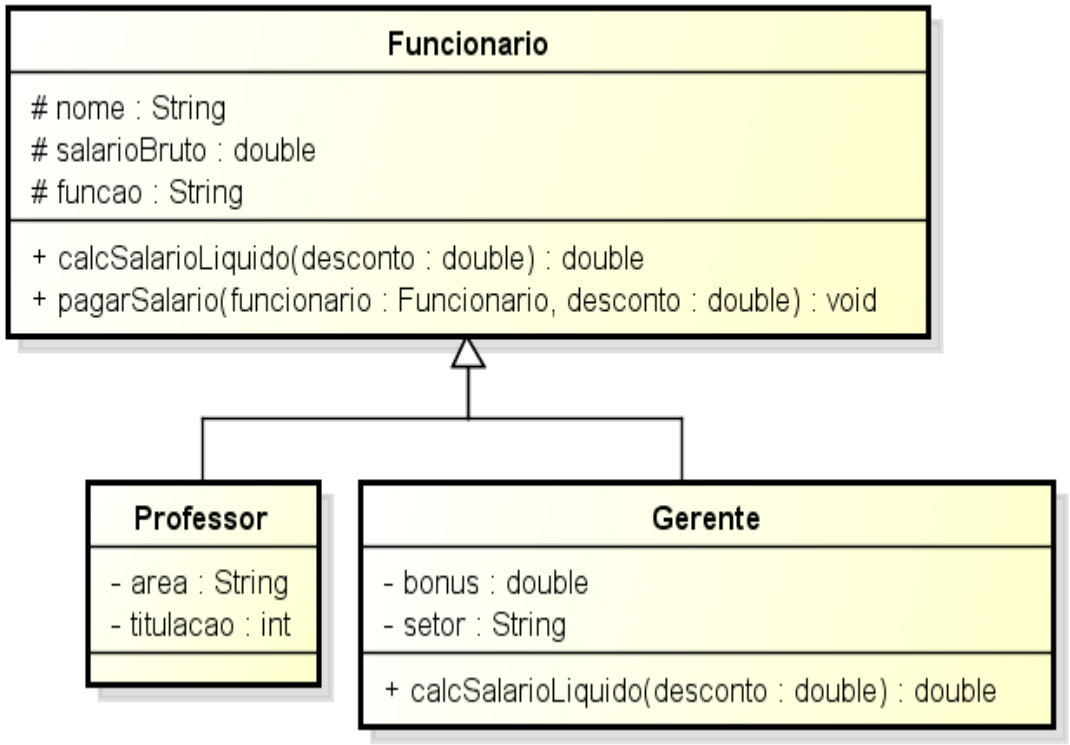
Exercício 12

- HERANÇA + Polimorfismo

Funcionario
calcSalarioLiquido = salarioBruto – desconto

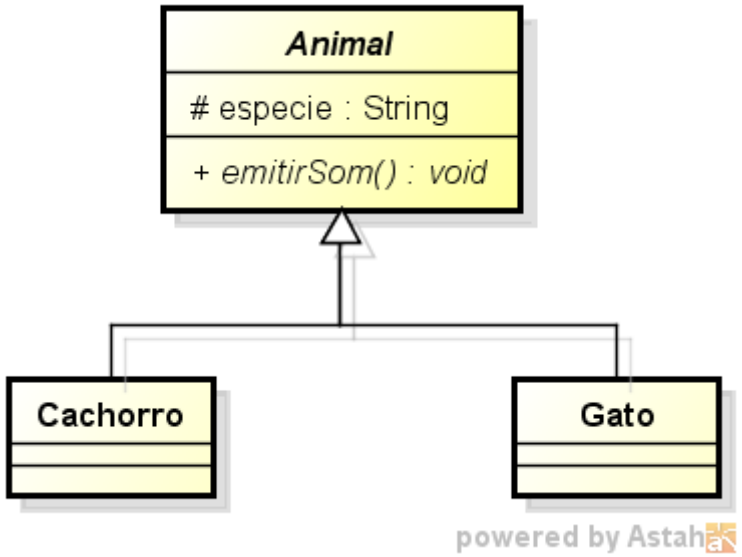
Gerente
calcSalarioLiquido = salarioBruto – desconto + bonus

Funcionario
pagarSalario = monta Msg: "Depositando {salarioLiquido} na conta de {nomeFuncionario}"



Polimorfismo – Classes Abstratas

- Conceito aplicado quando a classe for apenas um ponto de referência para as subclasses.
- Se a classe possuir um método abstrato ela obrigatoriamente deverá ser definida como abstrata.
- Classes abstratas não podem ser instanciadas



Polimorfismo – Classes Abstratas

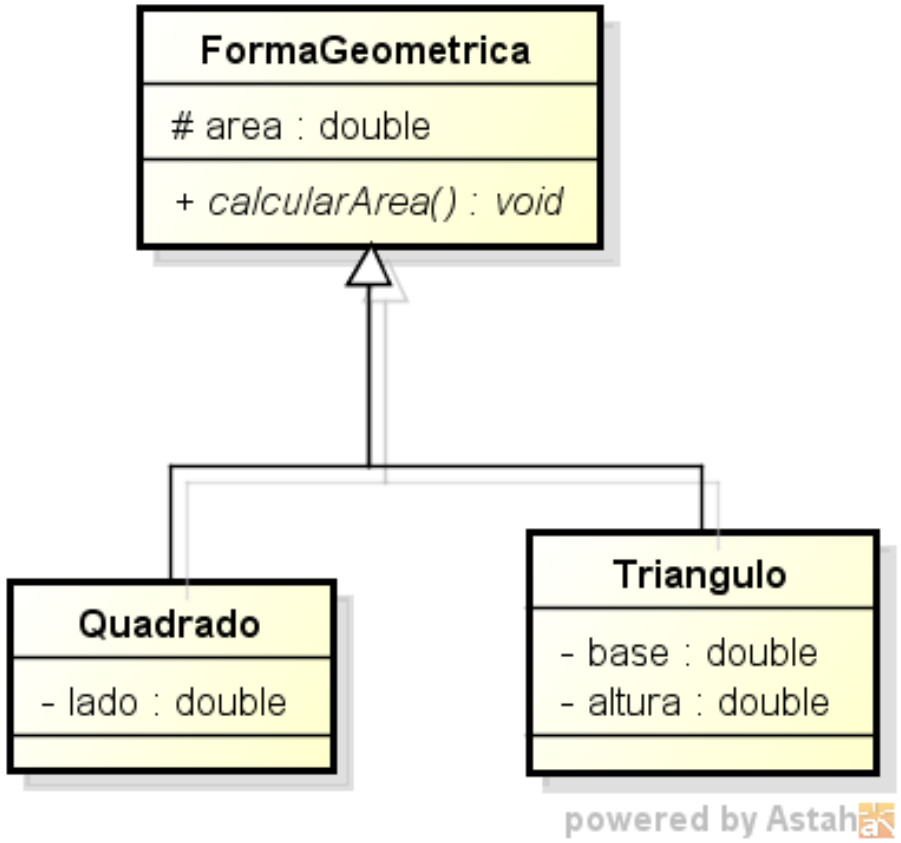
```
public abstract class Animal {

    protected String especie;

    abstract void emitirSom();

}
```

```
public class Cachorro extends Animal {

    void emitirSom() {
        System.err.println("Au au!!");
    }

}
```

```
public class Gato extends Animal{

    void emitirSom() {
        System.err.println("Miau miau!!");
    }

}
```

Exercício 13

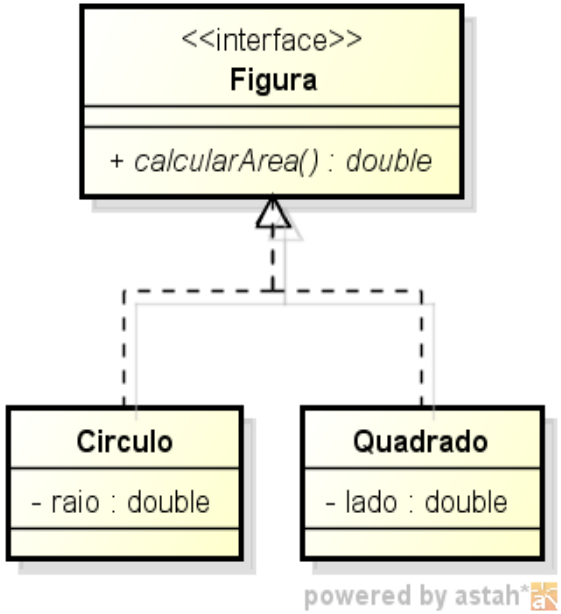
- HERANÇA + Polimorfismo (Abstract)



Interface

- As interfaces atuam como um **contrato** que define parte do comportamento de outras classes.
- Uma interface pode definir uma série de métodos, **mas nunca conter implementação deles**.
  - Expõe **o que o objeto deve fazer, e não como ele faz, nem o que ele tem.**
- As classes podem implementar mais de uma interface.

Interface




```
public interface Figura {
  public double calcularArea();
}
```

```
public class Circulo implements Figura {
  public double calcularArea() {
    //faz o cáculo da área do círculo
  }
}
```

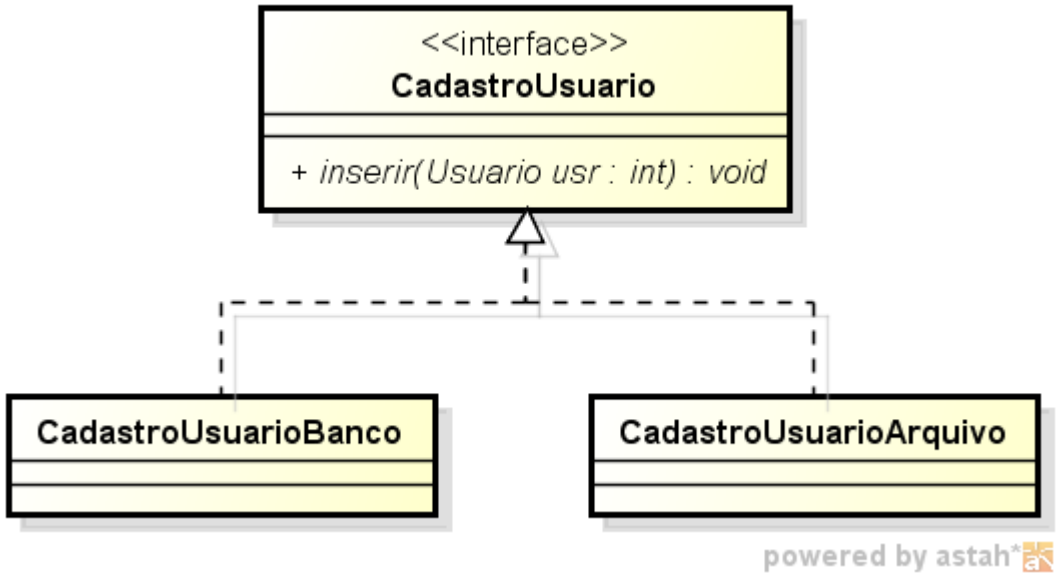
```
public class Quadrado implements Figura {
  public double calcularArea() {
    //faz cálculo da área do quadrado
  }
}
```

Interface

- Exemplo Classe de Execução
  - **ExecFigura**

```
Figura fig = new Circulo(10);
double area = fig.calcularArea();


fig = new Quadrado(8);
area = fig.calcularArea();
```

Interface




```
public interface CadastroUsuario {
  public void inserir( Usuario usr ) throws Exception;
}
```

```
public class CadastroUsuarioBanco implements CadastroUsuario {
  public void inserir( Usuario usr ) throws Exception {
    //insere os dados no banco
  }
}
```

```
public class CadastroUsuarioArquivo implements CadastroUsuario {
  public void inserir( Usuario usr ) throws Exception {
    //insere os dados no arquivo
  }
}
```

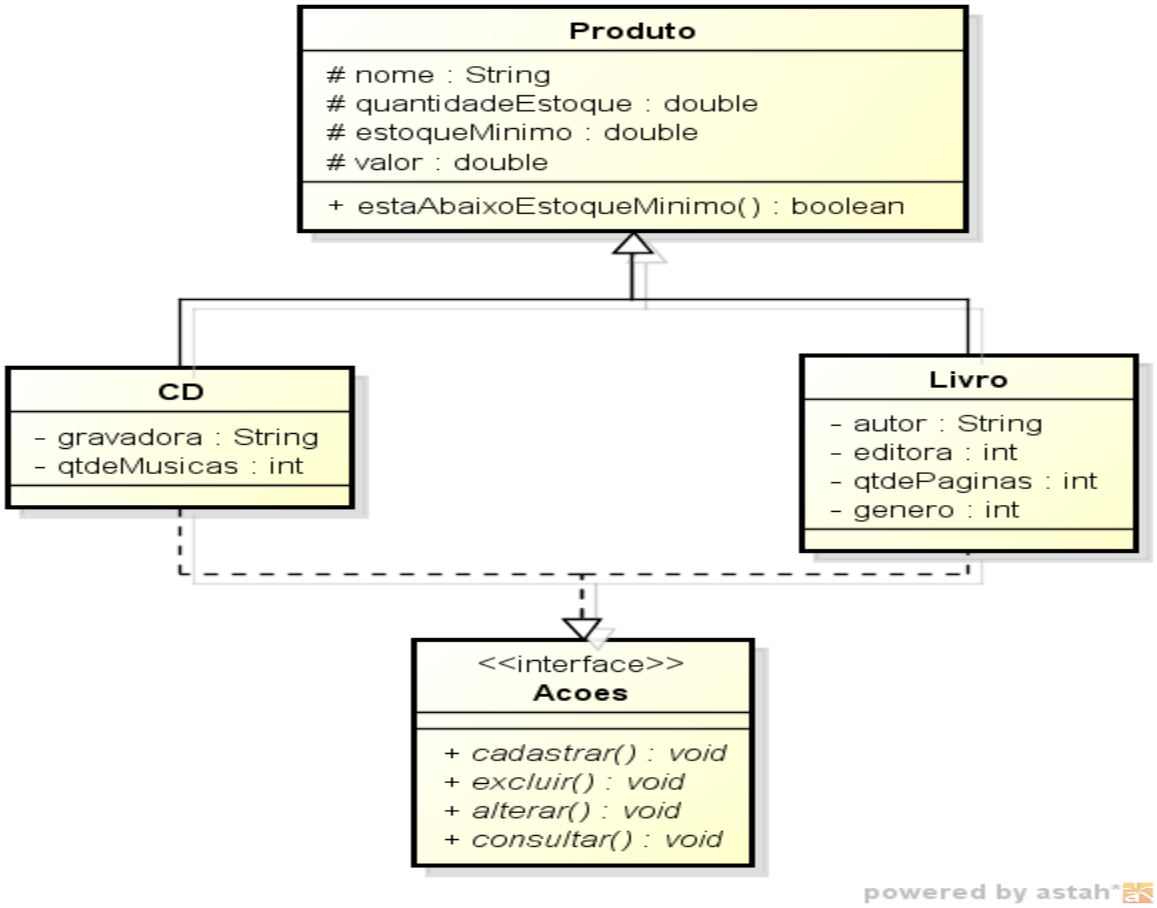
Interface

- Exemplo Classe de Execução
  - ExecCadastroUsuario

```
//...
CadastroUsuario cad = new CadastroUsuarioBanco();
cad.inserir( usuario );
cad = new CadastroUsuarioArquivo();
cad.inserir( usuario );
//...
```

Exercício 14

- Herança + Interface



Herança x Interface

Interface	Herança
Uma classe pode implementar mais de uma interface	Uma classe só pode ter uma ancestral.
Uma classe é obrigada a implementar TODOS os métodos da interface, caso contrário será considerada como ABSTRATA.	Uma classe já recebe a implementação de TODOS os métodos de sua ancestral. Ela pode, opcionalmente, fazer um Override dos métodos herdados.
Uma interface declara métodos e/ou constantes, sem implementação.	Uma classe ancestral pode ser totalmente funcional.
Métodos em uma interface não podem ser "private" ou "protected". Além disto os modificadores: "transient", "volatile" e "sinchronized" não podem ser utilizados.	Todos os modificadores podem ser utilizados em uma classe ancestral de outras.

Polimorfismo – Classes Abstratas

- Modificador: *abstract*

Tipo	Finalidade	Classe	Método	Atributo	Trecho Código
abstract	O elemento é virtual e **deve ser redefinido em sub-classes**. Se uma classe possuir um método declarado como “abstract”, ela deve ser declarada como “abstract”. **Você não pode implementar o método abstract em uma classe abstrata, mas deve implementá-lo em qualquer sub-classe.** A classe abstrata não pode ser instanciada	X	X	-----	------

Modificadores

Tipo	Finalidade	Classe	Método	Atributo	Trecho Código
Final	Significa que o elemento não pode ser alterado. Uma classe “final” não pode ter sub-classes (não pode ser herdada). Um método “final” não pode ser redefinido (sobrescrito) e uma variável “final” não pode ser alterada.	X	X	X	_____

Modificadores

- **Classe *final***
  - Uma classe “final” não pode possuir sub-classes. É o contrário de uma classe Abstrata.

```
public final class carro {
}

public class onibus extends carro {
}
```

Modificadores

- **Método *final***
- Um método “final” não pode ser redefinido (override). Redefinir um método é reescrevê-lo em sub-classes. Por exemplo:

```
public class veiculo {
     protected boolean ligado;
     public final boolean ligar() {
          ligado = true;
          return ligado;
     }
}
```

```
public class carro extends veiculo {
     public boolean ligar() {
          ligado = true;
          return ligado;
     }
}
```

Modificadores

- **Variáveis *final***
  - São atributos de uma classe que não mudam de valor. O modificador ***final*** indica que o atributo é imutável

```
public class Constante {
    static final double PI = 3.14159265;
}
```

```
class teste {
    public static void main( String[] args ) {
        System.out.println(Constante.PI);
        Constante.PI = 0;
    }
}
```

Modificadores

Tipo	Finalidade	Classe	Método	Atributo	Trecho Código
Static	Se aplicado a uma variável, significa que ela pertence à classe e não à instância do Objeto. Se aplicado a um método, este passa a ser também da Classe e somente pode acessar suas variáveis Estáticas. Se aplicado a um trecho de código, este será executado no momento em que a Classe for carregada pelo JVM.	_____	X	X	X

Modificadores

- ***Static***
  - Atributos Estáticos
    - Atributos estáticos não precisam de uma instância da classe para serem usados.
      - Eles podem ser acessados diretamente
    - Eles são compartilhados por todas as instâncias da classe (cuidado ao usá-los)
      - Como se fossem variáveis globais

Modificadores

- ***Static***
  - Atributos Estáticos

```
class Contador {
    static int count = 0;
    void incrementar() {
    count++;
    }
}
```

```
class TestandoContador_1{
    public static void main( String[] args ) {
        System.out.println("Contador: " + Contador.count);
        System.out.println(Contador.count++);

        Contador c1 = new Contador();
        System.out.println(c1.count);

        Contador c2 = new Contador();
        System.out.println(c2.count);

        }
}
```

Modificadores

- ***Static***
  - Métodos Estáticos
    - Não precisam de uma instância da classe para serem usados .
    - Métodos estáticos NÃO podem chamar métodos não-estáticos sem uma instância.

Modificadores

- ***Static***

```
class MetodoEstatico {
    public static void main( String[] args ) {
        MetodoEstatico me = new MetodoEstatico();
        me.metodoNaoEstatico();
        me.metodoEstatico();
        MetodoEstatico.metodoEstatico();
        metodoEstatico();
    }

    static void metodoEstatico() {
        //metodoNaoEstatico(); //ERRADO
        // (new MetodoEstatico()).metodoNaoEstatico();  //OK
    }

    void metodoNaoEstatico() {
        metodoEstatico();  //OK
    }
}
```