Apostila
Técnicas de Orientação a Objetos

Etec Zona Leste

Professor Wagner Lucca

Índice:

Introdução à Orientação a Objetos:

No mundo em que vivemos, de constante utilização e atualização de recursos tecnológicos, os softwares ganharam um espaço que vem aumentando vertiginosamente e em diversos setores da sociedade. Existem programas específicos para uso comercial, industrial, administrativo, financeiro, governamental, militar, científico, de entretenimento etc.

Para atender a essa demanda cada vez maior e mais exigente, as áreas de desenvolvimento e manutenção de softwares precisam criar aplicações cada vez mais complexas.

A orientação a objetos surgiu da necessidade de elaborar programas mais independentes, de forma ágil, segura e descentralizada, permitindo que equipes de programadores localizadas em qualquer parte do mundo trabalhem no mesmo projeto.

Essa técnica de desenvolvimento de softwares, mais próxima do ponto de vista humano, torna mais simples e natural o processo de análise de problemas cotidianos e a construção de aplicações para solucioná-los.

Para que isso seja possível, é preciso quebrar paradigmas em relação ao modelo procedural, que tem como base a execução de rotinas ou funções sequenciadas e ordenadas para atender aos requisitos funcionais de uma aplicação.

Nele, as regras (codificação) ficam separadas dos dados (informações processadas).

Por isso, o programador passou a estruturar a aplicação para que, além de resolver problemas para os quais foi desenvolvido, o software possa interligar elementos como programação, banco de dados, conectividade em rede, gerenciamento de memória, segurança etc.

Isso aumenta significativamente a complexidade na elaboração e expansão desses softwares.

O modelo orientado a objetos tem como base a execução de métodos (pequenas funções) que atuam diretamente sobre os dados

de um objeto, levando em conta o modo como o usuário enxerga os elementos tratados no mundo real.

Nessa técnica, o desenvolvedor cria uma representação do que se pretende gerenciar no sistema (um cliente, um produto, um funcionário, por exemplo) exatamente como acontece no mundo real.

Assim, esses elementos podem interagir, trocando informações e adotando ações que definem os processos a serem gerenciados pelo sistema.

Por exemplo, no processo de venda de uma empresa podem ocorrer os seguintes passos: um cliente compra um ou vários produtos; o cliente é atendido por um vendedor; o vendedor tem direito a comissão.

Ao analisarmos esses processos no mundo real, destacamos como "entidades" envolvidas os seguintes elementos: cliente, vendedor e produto, assim como a venda e o cálculo e comissão.

A partir da análise orientada a objetos, representamos essas entidades dentro da nossa aplicação com as mesmas características adotadas no mundo real (nome, endereço e telefone de um cliente, etc. além de suas atitudes típicas numa transação comercial (cotação, compra, pagamento etc.).

Resumindo vantagens da orientação a objeto:

Produção de software mais confiável:

- proteção aos dados: encapsulamento;
- menor chance de resultados inesperados;

Aumento da produtividade de software:

- reutilização de código: classes;

Abstração:

Abstração é a capacidade do ser humano de se concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais.

Outra definição: habilidade mental que permite visualizar os problemas do mundo real com vários graus de detalhe, dependendo do contexto do problema.

Para o programador que utiliza orientação a objetos, a abstração é a habilidade de extrair do mundo real os elementos (entidades, características, comportamentos, procedimentos) que realmente interessam para a aplicação desenvolvida.

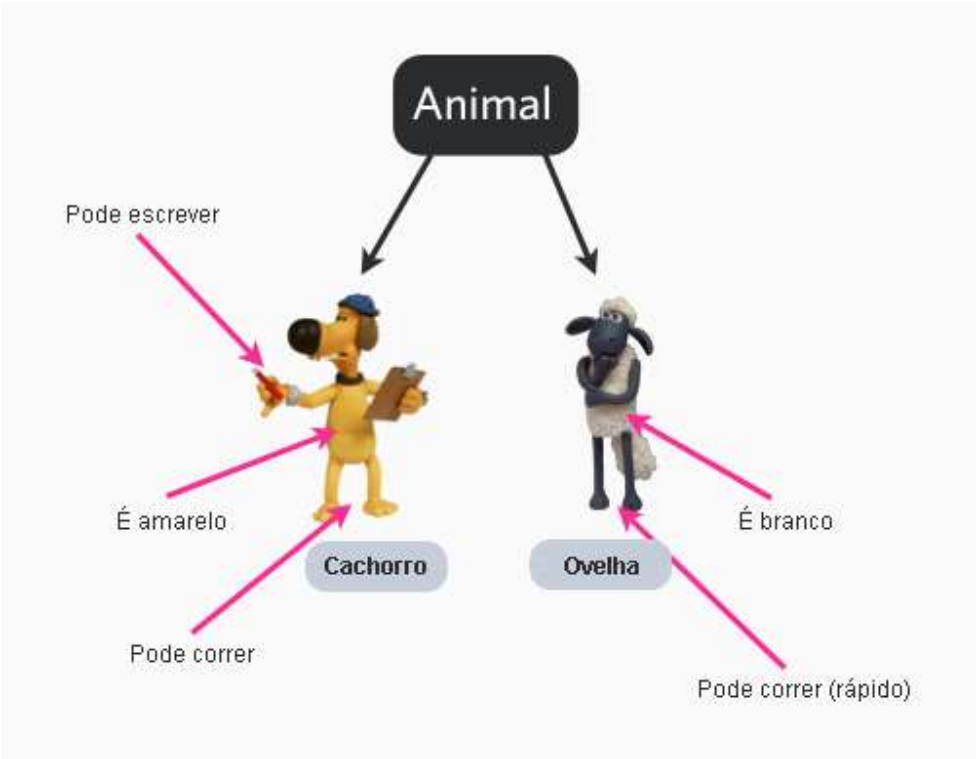
Por exemplo, as informações relevantes a respeito de um CD para uma aplicação usada no gerenciamento de uma loja que vende CDs, são diferentes das de uma aplicação para o gerenciamento de uma fábrica, que produz CDs.

Nota : Duas ou mais pessoas podem ver características distintas em um mesmo objeto. O objeto comporta ambas as características porém cada uma das pessoas visualiza aquilo que mais atendem a sua realidade, mediante aos seus próprios filtros internos. Isso chama-se **abstração.**

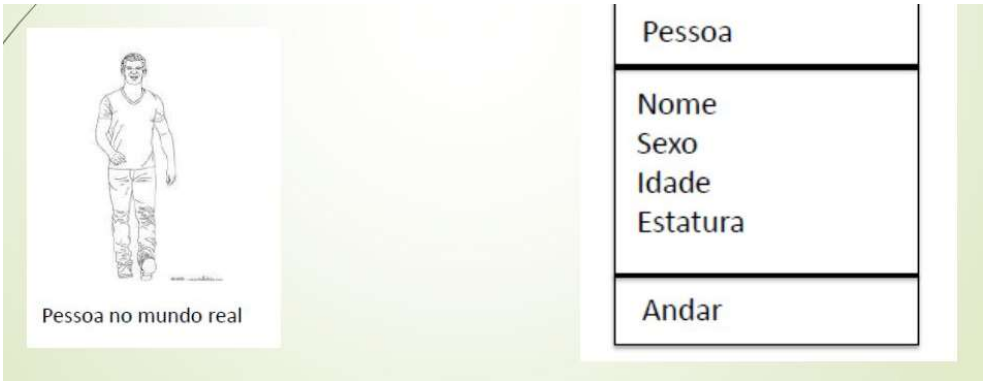
Objetos:

Um objeto representa um elemento que pode ser identificado de maneira "única";

Praticamente tudo pode ser considerado um objeto;



Veja outro exemplo:



Classes, atributos e métodos:

Classe é uma representação de um tipo de objeto.

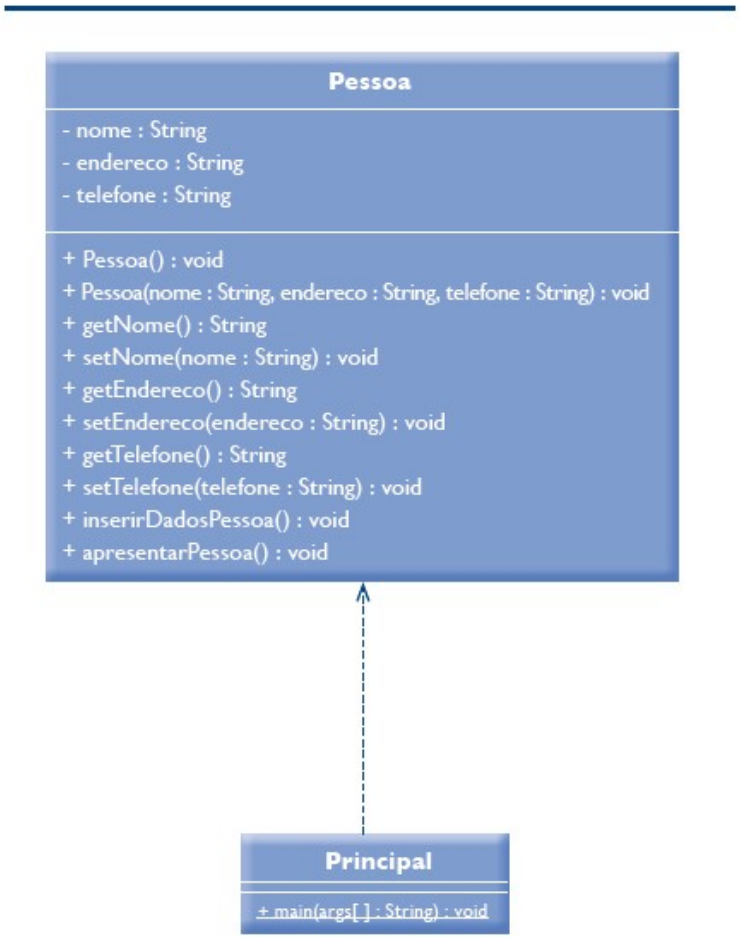
Atributo são suas características.

Em outras palavras são as informações (dados) que definem as características e os valores pertinentes à classe e que serão armazenados nos (futuros) objetos são chamadas de atributos.

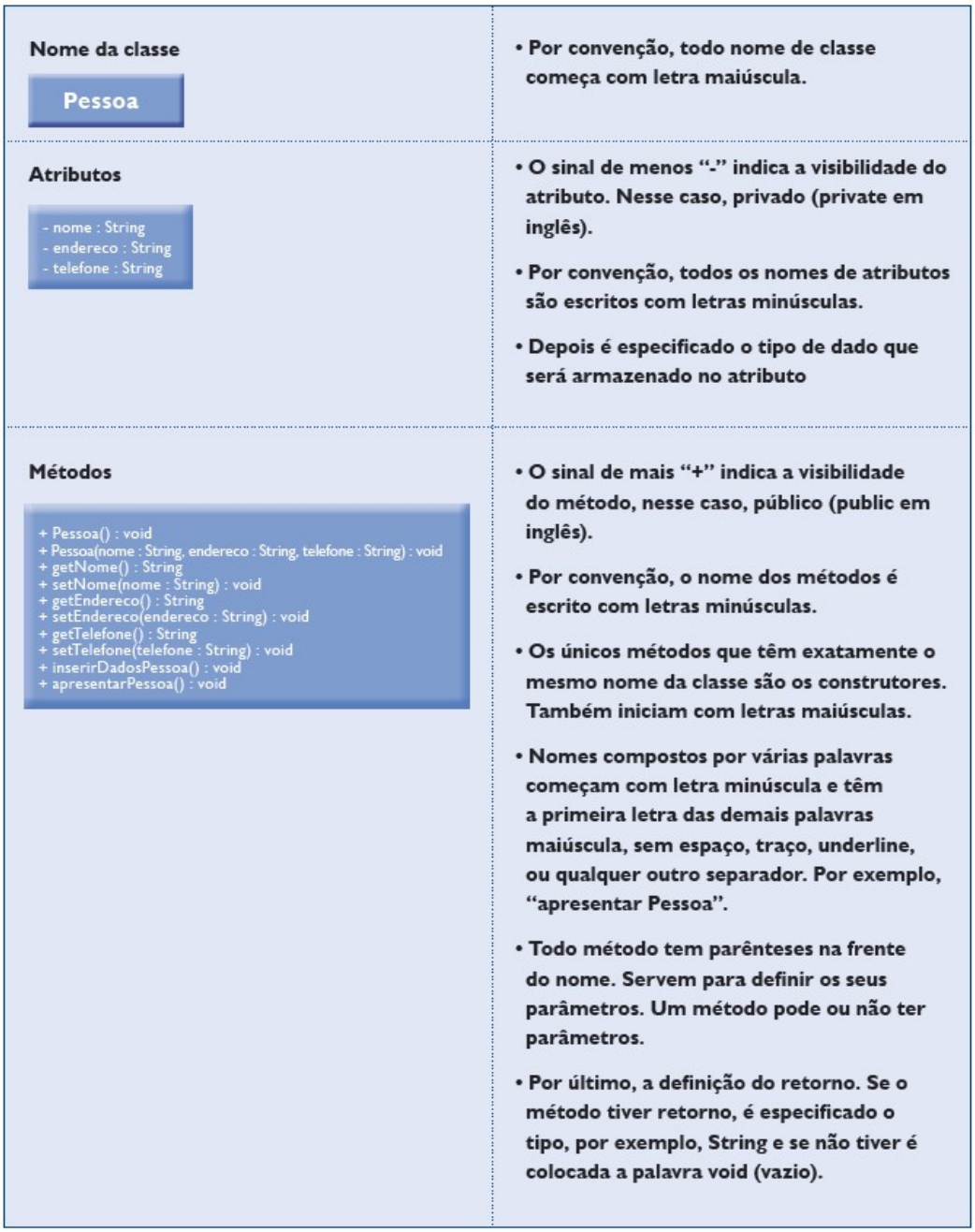
Métodos podem modificar seus atributos, podem expressam ações.

Métodos são blocos de código que pertencem a uma classe e têm por finalidade realizar tarefas. Geralmente, correspondem a uma ação do objeto. Considerando-se o exemplo do automóvel, os métodos poderiam definir processos como ligar, acelerar, frear etc.

Observe o exemplo:



Classe pessoa em detalhes:



Em código:

Pessoa.java

```
1  public class Pessoa {
2
3      // Atributos
4      private String nome;
5      private String endereco;
6      private String telefone;
7
8      // Método construtor inicializando os atributos sem valores
9      public Pessoa() {
12
13     // Método construtor inicializando os atributos com valores passados por parâmetros
14     public Pessoa(String nome, String endereco, String telefone) {
19
20     // Métodos de acesso (getters e setters)
21
22     // Retorna o conteúdo do atributo nome
23     public String getNome() {
26
27     // Altera o conteúdo do atributo nome
28     public void setNome(String nome) {
31
32     public String getEndereco() {
35
36     public void setEndereco(String endereco) {
39
40     public String getTelefone() {
43
44     public void setTelefone(String telefone) {
47
48     // Métodos específicos da classe
49
50     // Apresenta o conteúdo dos atributos
51     public void apresentarPessoa(){
56
57 }
```

Detalhes do método apresentarPessoa():

```
50      // Apresenta o conteúdo dos atributos
51      public void apresentarPessoa(){
52          System.out.println("Nome: " + this.getNome());
53          System.out.println("Endereço: " + this.getEndereco());
54          System.out.println("Telefone: " + this.getTelefone());
55      }
```

Outros conceitos importantes:

Classes de modelagem: são as que darão origem a objetos. Ou, ainda, são as que definem novos (e personalizados) "tipos de dados".

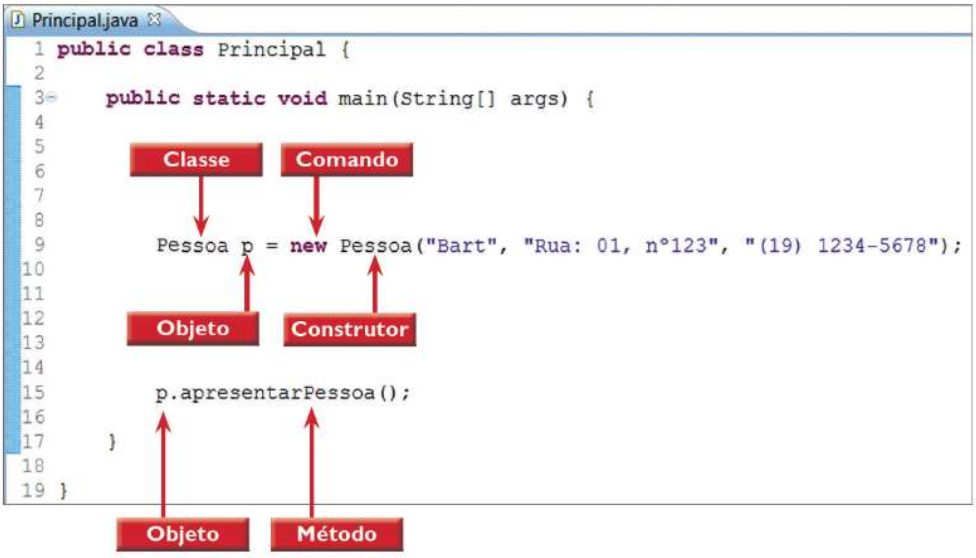
Classe Principal: terá, por enquanto, a finalidade de implementar o método main (principal). A classe principal apenas organiza a sequência das execuções (solicitações) dos

métodos que estão definidos nas classes de modelagem. A classe Principal que possui o método main não executada nada, ela é um maestro que organiza a sequência dos métodos.

Instanciar um objeto:

Significa alocar um objeto na memória do computador. Observe o exemplo:

Quando declaramos Pessoa p = new Pessoa(), estamos alocando um objeto do tipo Pessoa na memória.



Modificadores de acesso:

Tanto os atributos quanto os métodos devem (por clareza de código) ter sua visibilidade definida na declaração (para atributos) e na assinatura (para os métodos).

Visibilidade, no caso, significa deixá-los ou não disponíveis para uso em outras classes da aplicação.

Nas classes de modelagem, a visibilidade define se determinado atributo ou método ficará acessível por intermédio do objeto depois de instanciado.

Por exemplo, o método apresentarPessoa foi chamado (acessado) a partir do objeto p (p.apresentarPessoa()) na classe Principal, graças ao fato de ter sido definido como public na classe Pessoa.

Se tentarmos acessar, por exemplo, o atributo email do objeto, não será possível.

Na verdade, esse atributo não fica visível na classe Principal, porque foi definido como private na classe Pessoa.

O uso de modificadores de acesso não é obrigatório.

Se forem omitidos, a visibilidade default (padrão) adotada é a protected (protegido) – visível dentro do mesmo pacote.

Porém, seu uso deixa o código mais claro em relação ao que está acessível ou não por intermédio dos futuros objetos.

Private:

Os atributos e métodos definidos por esse modificador somente são visíveis (podem ser acessados) dentro da classe em que foram codificados.

Por exemplo, o atributo e-mail definido na classe Pessoa.

Dentro dela (normalmente nos métodos da classe) esse atributo pode ser manuseado (seu valor lido ou alterado).

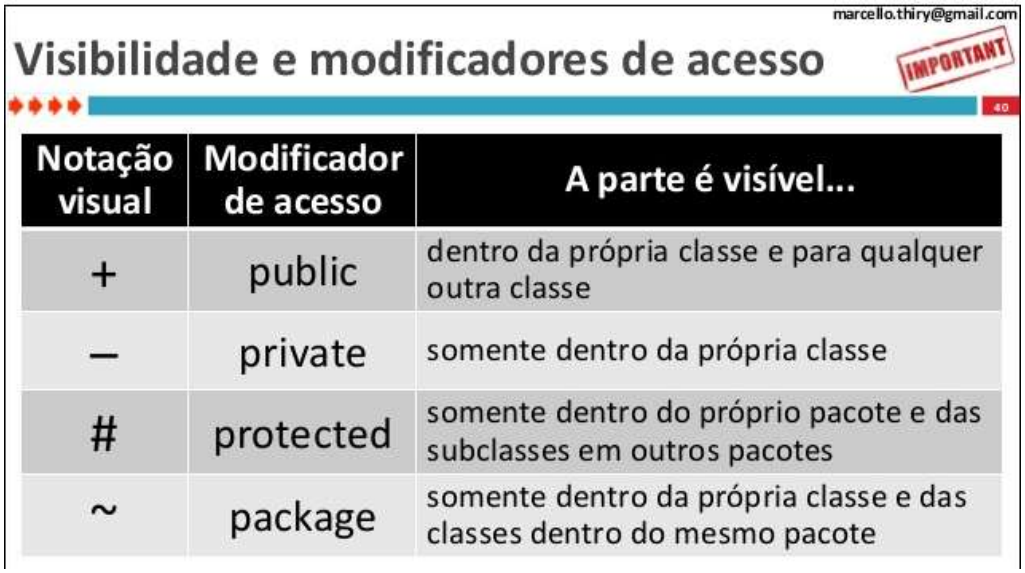
Public:

Os atributos e métodos definidos com o modificador public são visíveis em qualquer lugar dentro da classe e podem ser acessados sem nenhuma restrição a partir de um objeto instanciado.

Protected:

Um terceiro tipo, o protected (protegido, em português), define que um atributo ou método pode ser visto dentro da própria classe e em objetos instanciados a partir de classes que pertençam ao mesmo pacote (package em inglês).

Resumo:



Notação visual	Modificador de acesso	A parte é visível...
+	public	dentro da própria classe e para qualquer outra classe
–	private	somente dentro da própria classe
#	protected	somente dentro do próprio pacote e das subclasses em outros pacotes
~	package	somente dentro da própria classe e das classes dentro do mesmo pacote

Métodos construtores:

Outra definição que busca maior clareza de código é a implementação do construtor, um método especial responsável pela inicialização dos atributos de um objeto no momento de sua criação (instanciação).

Se sua codificação for omitida, é acionado o construtor default (padrão), que inicializa os atributos sem nenhum conteúdo.

Em nossos exemplos, adotaremos a prática de definir pelo menos dois construtores por classe: um que inicializará os atributos vazios e outro que possibilitará a passagem de valores a serem armazenados nos atributos.

Uma classe pode ter quantos construtores forem necessários para aplicação.

E todos eles terão sempre o mesmo nome da classe (inclusive com a primeira letra maiúscula).

Porém, com a passagem de parâmetros diferentes.

Os construtores estão diretamente relacionados aos atributos.

Portanto, se uma classe não tiver atributos, os construtores serão desnecessários.

A codificação dos construtores na classe Pessoa ficará como ilustra a figura.

```
public Pessoa() {
    this("","","");
}

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

O primeiro construtor, sem parâmetros (nada entre os parênteses), adiciona "vazio" em cada um dos atributos (porque os três atributos são String).

Já o segundo está preparado para receber três parâmetros, que são os valores do tipo String, identificados como nome, endereco e telefone (lembre-se de que a referência é os valores dentro dos parênteses).

Esses valores serão atribuídos (armazenados) na sequência em que estão escritos (o primeiro parâmetro no primeiro atributo, o segundo parâmetro no segundo atributo e o terceiro parâmetro no terceiro atributo).

O comando this:

O this é utilizado como uma referência ao próprio objeto que está chamando um método.

O uso desse comando também deixa claro quando estamos nos referindo a um atributo ou método da mesma classe na qual estamos programando.

Não é obrigatório, porém, dentro do construtor que recebe parâmetros, por exemplo, é possível visualizar o que é parâmetro e o que é atributo.

Isso acontece, apesar de os dois terem exatamente os mesmos nomes (os atributos vêm precedidos do this).

Métodos de acesso:

Para garantir o encapsulamento e possibilitar o acesso aos dados do objeto, são criados métodos de acesso aos atributos em casos específicos.

Esses métodos são codificados dentro da classe e, por isso, definidos com acessibilidade pública (public).

Significa que eles podem ser acessados a partir do objeto instanciado de qualquer classe.

Servem, portanto, como portas de entrada e de saída de informações dos atributos, por onde seus valores podem ser lidos e alterados.

A vantagem de abrir esse acesso via método é poder definir (programar) as regras de leitura e escrita nesses atributos.

Permite, como no exemplo da loja, identificar se o processo que está "solicitando" o conteúdo do atributo estoqueDisponível é autorizado a fazer tal consulta.

Ou, mais crítico ainda, se permite atualizar (alterar) esse atributo depois da venda efetuada (subtraindo a quantidade vendida), liberar a leitura do atributo limiteDeCredito e bloquear a sua escrita (alteração).

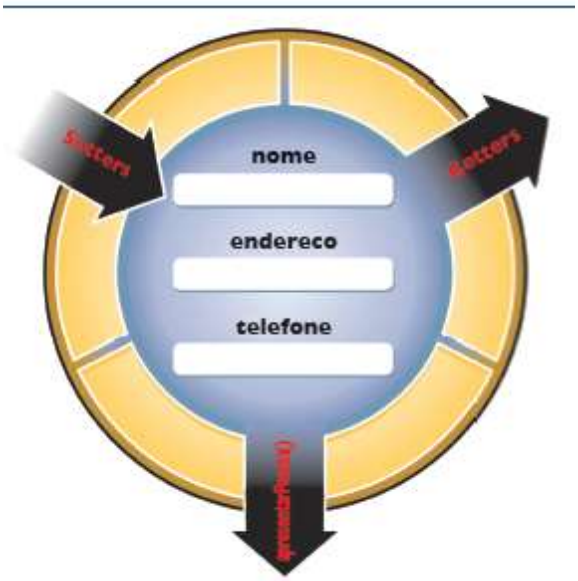
O cálculo do limite fica a cargo do método responsável por essa operação.

Resumindo nenhuma outra classe ou objeto pode acessar diretamente os atributos encapsulados.

Se isso for necessário, deve-se "pedir" ao objeto para que ele mostre ou altere o valor de um de seus atributos mediante suas próprias regras (definidas nos métodos de acesso).

Apesar de não ser obrigatório, adotaremos a definição de um método de leitura e de um método de escrita para cada atributo da classe.

Voltando ao exemplo da classe Pessoa, para acessar o atributo nome, temos os seguintes métodos de acesso: get e set.



Método get:

Por padrão do próprio Java, os métodos que leem e mostram (retornam) o conteúdo de um atributo começam com a palavra get, seguida pelo nome desse atributo.

```
public String getNome() {
    return nome;
}
```

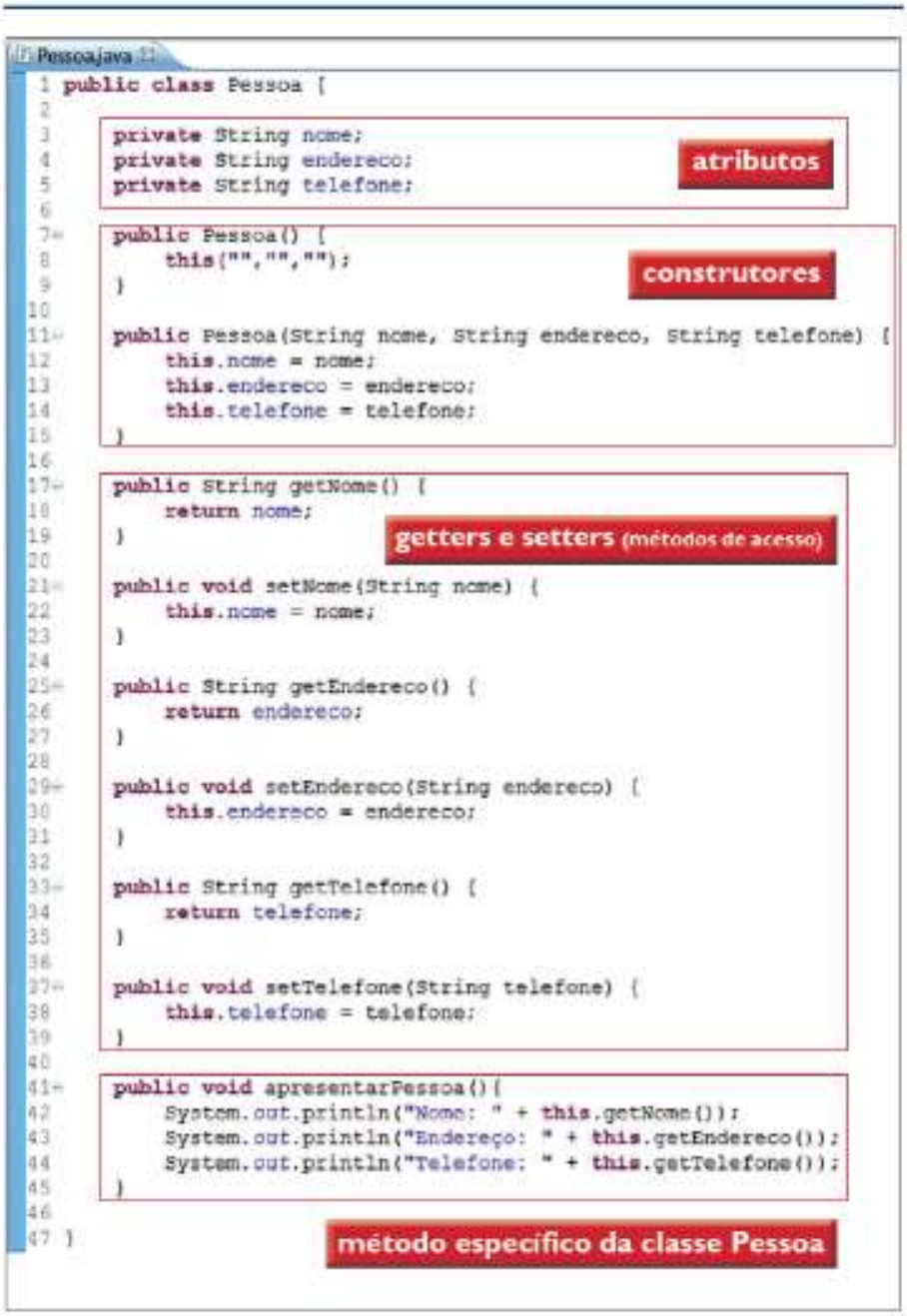
Método set:

Já os métodos responsáveis por escrever (alterar) o conteúdo de um atributo começam com a palavra set, seguida pelo nome desse atributo.

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

Visão geral da classe Pessoa e sua estrutura:

A figura reproduz a codificação completa da classe Pessoa, destacando as camadas vistas até agora.




```
1  public class Pessoa {
2
3      private String nome;
4      private String endereco;
5      private String telefone;
6
7      public Pessoa() {
8          this("","","");
9      }
10
11     public Pessoa(String nome, String endereco, String telefone) {
12         this.nome = nome;
13         this.endereco = endereco;
14         this.telefone = telefone;
15     }
16
17     public String getNome() {
18         return nome;
19     }
20
21     public void setNome(String nome) {
22         this.nome = nome;
23     }
24
25     public String getEndereco() {
26         return endereco;
27     }
28
29     public void setEndereco(String endereco) {
30         this.endereco = endereco;
31     }
32
33     public String getTelefone() {
34         return telefone;
35     }
36
37     public void setTelefone(String telefone) {
38         this.telefone = telefone;
39     }
40
41     public void apresentarPessoa(){
42         System.out.println("Nome: " + this.getNome());
43         System.out.println("Endereço: " + this.getEndereco());
44         System.out.println("Telefone: " + this.getTelefone());
45     }
46
47 }
```

Herança:

A herança é um conceito amplamente utilizado em linguagens orientadas a objetos.

Além de vantagens facilmente identificadas, como a reutilização e organização de códigos, a herança também é a base para outros conceitos, como a sobrescrita de métodos, classes e métodos abstratos e polimorfismo.

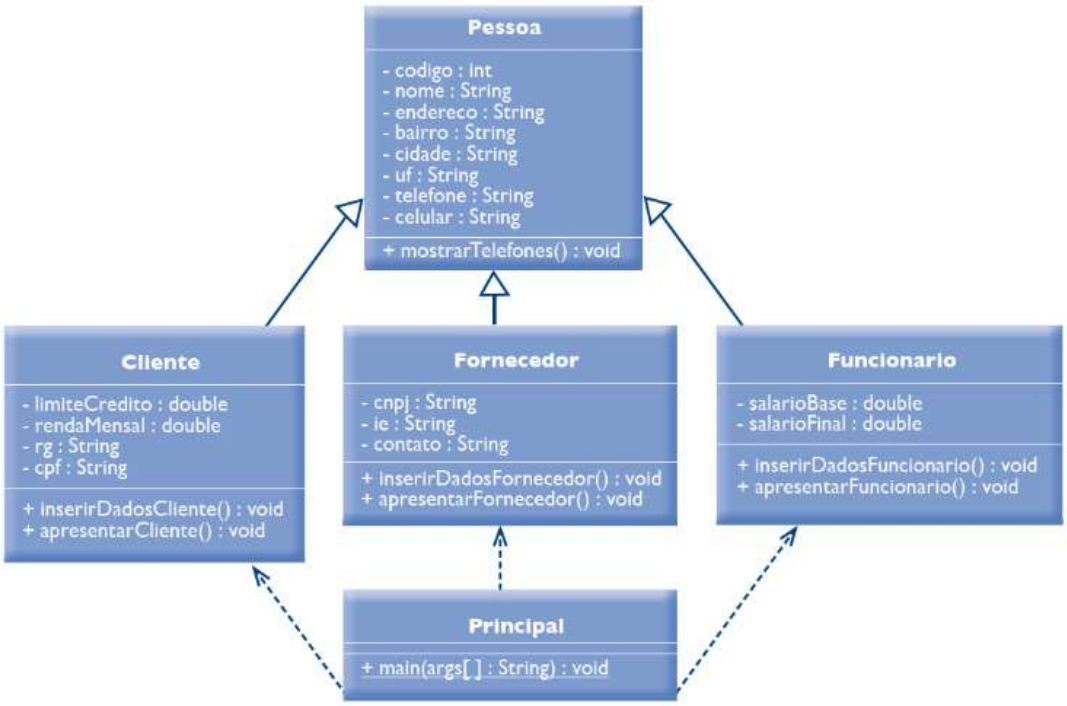
Tais conceitos são fundamentais para a modelagem de sistemas mais robustos.

Durante a análise dos requisitos de um sistema, podemos destacar os atributos ou os métodos comuns a um grupo de classes e concentrá-los em uma única classe (processo conhecido como generalização).

Chamamos de superclasses essas classes que concentram atributos e métodos comuns que podem ser reutilizados (herdados) e de subclasses aquelas que reaproveitam (herdam) esses recursos.

Observação: Podemos também adotar a nomenclatura de classe pai (superclasse) e classe filha (subclasse).

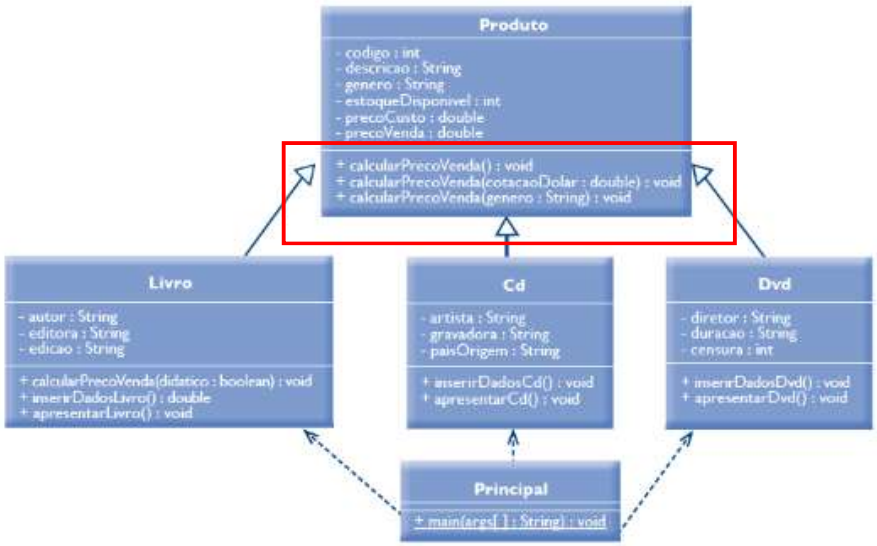
Observe as definições de clientes, fornecedores e funcionários no diagrama de classes a seguir.



Sobrecarga de método (overload):

A programação em sistemas desenvolvidos com Java é distribuída e organizada em métodos.
Muitas vezes, os programadores se deparam com situações em que um método deve ser usado para finalidades semelhantes, mas com dados diferentes.

Por exemplo, os produtos comercializados na livraria são livros, CDs e DVDs, como representado no diagrama de classe a seguir.



O cálculo do preço de venda dos produtos da livraria depende de alguns fatores: em determinadas épocas do ano, todos os produtos recebem a mesma porcentagem de acréscimo em relação ao preço de custo.

Se o produto em questão for importado, é necessário considerar a cotação do dólar.

Em algumas situações (promoções, por exemplo), é preciso atualizar todos os produtos de um determinado um gênero.
E, no caso específico dos livros didáticos, o cálculo do preço de venda é diferente dos demais.

Em outras linguagens de programação, como não é possível termos duas funções (blocos de código equivalentes a métodos) com o mesmo nome, nos depararíamos com a necessidade de criar funções nomeadas (calcularPrecoVenda1, calcularPrecoVenda2, calcularPrecoVenda3, calcularPrecoVenda4 ou calcularPrecoVendaNormal, calcularPrecoVendaImportado, calcularPrecoVendaPorGenero e calcularPrecoVendaLivroDidatico e assim sucessivamente).

Dessa forma, além de nomes extensos, e muitas vezes estranhos, teríamos uma quantidade bem maior de nomes de funções para documentar no sistema.

Em Java, para situações desse tipo, usamos a sobrecarga que considera a identificação do método pela assinatura e não somente pelo nome.

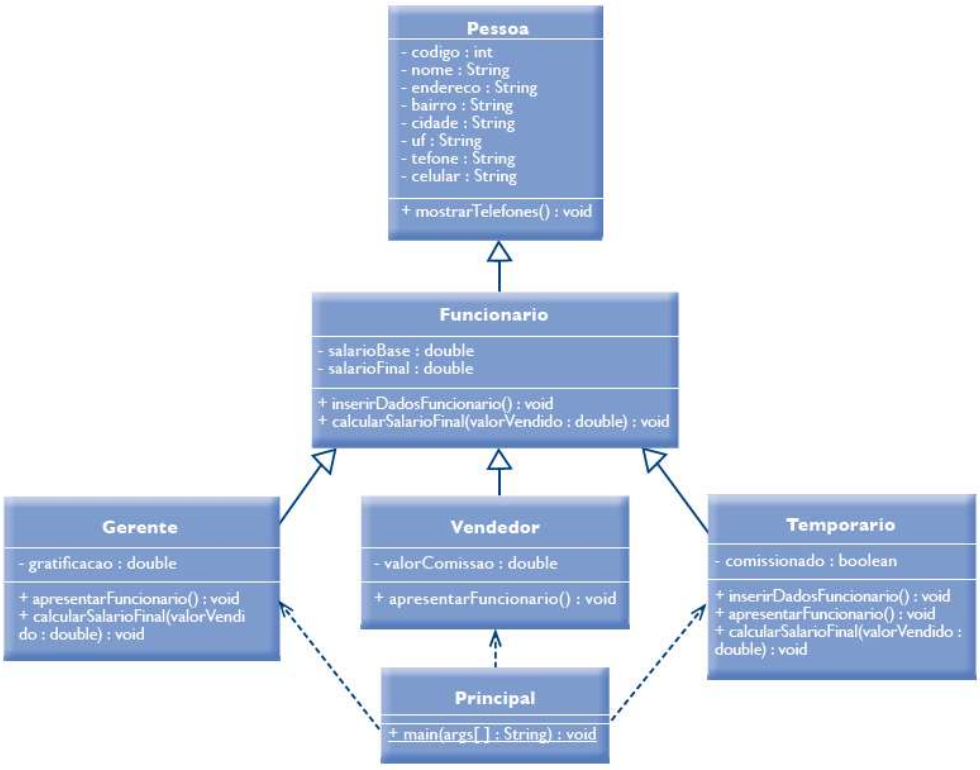
Como já vimos, a assinatura de um método é composta pelo nome e pela passagem de parâmetros.

Assim, é possível definir os métodos com o mesmo nome (calcularPrecoVenda, como no diagrama) e alternar a passagem de parâmetros.

Sobrescrita de método (override):

A sobrescrita de métodos está diretamente relacionada com a herança e consiste em reescrever um método herdado, mudando seu comportamento, mas mantendo exatamente a mesma assinatura.

Para exemplificar, utilizaremos o cálculo do salário final dos funcionários de uma livraria.



Na superclasse Funcionario, foi implementado o método calcularSalarioFinal considerando uma regra geral de cálculo (somar 10% do valor vendido ao salário base).

Consequentemente, esse método foi herdado por suas subclasses Gerente, Vendedor e Temporario.

O problema é que, de acordo com as normas da livraria, o cálculo do salário dos gerentes e dos temporários é diferente.

Para os vendedores, o método calularSalarioFinal herdado está correto, porém, para Gerente e Temporario, não.

O problema pode ser solucionado com a sobrescrita dos métodos calcularSalarioFinal nas classes Gerente e Temporario. Para tal, basta implementar o método mantendo a assinatura, e uma linha antes adicionar a marcação @Override

```
@Override
public void calcularSalarioFinal (Double valorVendido)
{
    // Modifique a lógica do método conforme necessidade;
}
```

Classes e métodos abstratos:

A classe Abstrata ou do inglês Abstract é uma classe modelo para uma implementação futura, ela não precisa ter o código definido em seus métodos mas serve de base para a construção de outra classe no futuro. Observe o exemplo:

```
// Classe Carro;
abstract class Carro
{
    // Todo carro tem placa e cor;
    abstract void cor();
    abstract void placa();
}
```

```
// Agora podemos criar nossas classe "Concretas" com base na
nossa classe abstrata:

// Classe Ferrari;
class Ferrari extends Carro
{
     void cor()
     {
          System.out.println("vermelho");
     }

     void placa()
     {
          System.out.println("JHKL1025");
     }
}

// Outra classe, Celta;
class Celta extends Carro
{
     void cor()
     {
          System.out.println("Rosa");
     }

     void placa()
     {
          System.out.println("HJU677");
     }
}
```

Polimorfismo:

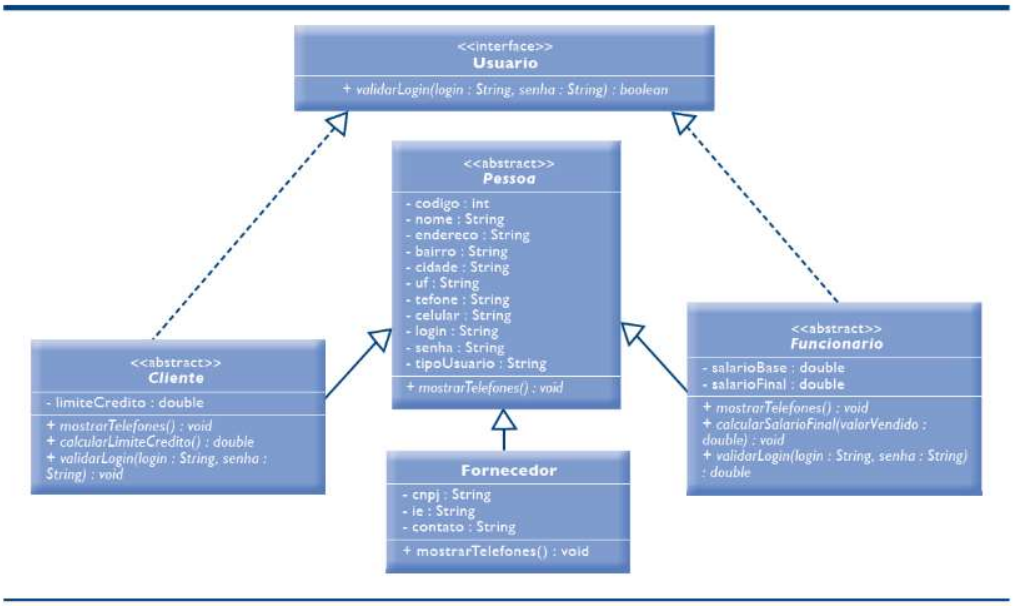
Polimorfismo é a capacidade de uma referência de um tipo genérico referenciar um objeto de um tipo mais especifico, exemplo:

```
public interface Carro
{
   public void acelerar();
}


public Ferrari implements Carro
{
   public void acelerar()
   {
      System.out.println("Ferrari acelerando...");
   }
}

public Fusca implemets Carro
{
   public void acelerar()
   {
      System.out.println("Fusca tentando acelerar...");
   }
}

Carro c = new Ferrari();
c.acelerar();

c = new Fusca();
c.acelerar();
```

Interfaces:

Ainda no nível da modelagem do sistema, as interfaces são tipos especiais de classes que servem para definir padrões de como determinados grupos de classes poderão ser usados, definindo assinaturas de métodos pelas classes que deverão ser adotados obrigatoriamente pelas classes (e subclasses) que as implementarem.

Uma interface é composta somente por métodos abstratos (não possui atributos nem métodos concretos) e pode ser vista como uma espécie de "contrato", cujas especificações das classes que as "assinam" (implementam) se comprometem a seguir, ou seja, devem seguir os métodos nela definidos.



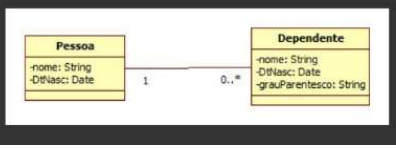
Relação de Objetos:

No mundo real os objetos se relacionam um com os outros, por exemplo não existe carro, sem motor ou pedido sem item. Da mesma maneira podemos relacionar objetos através da orientação a objetos.

Associação:

ASSOCIAÇÃO é o mecanismo pelo qual um objecto utiliza os recursos de outro. Pode tratar-se de uma associação simples "usa um" ou de um acoplamento "parte de".

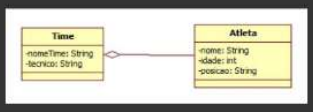
Vejamos um exemplo através do Diagrama de Classes, e logo após através do Código:




```
public class Pessoa {

private String nome;
private date dtNasc;
private List <Dependente> dependentes =
new ArrayList <Dependente> (0);

public void setDependentes
(list<Dependente> dependentes) {
}
public List<Dependente> getDependentes {
}
public void addDependente (Dependente de dependente) {
this.dependentes.add(dependente);

}

}

Public class Dependente {

private String nome;
private Date dtnasc;
private String grauParentesco;
private Pessoa responsavel;

public void setResponsavel (pessoa resp) {

}
public Pessoa getResponsavel() {

}

}
```

Agregação:

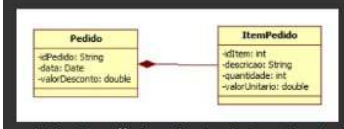
A **AGREGAÇÃO** indica que uma das classes do relacionamento é uma parte ou está contida em outra classe. Semanticamente representa: "consiste em", "contém", "é parte de".




```
public class Atleta {
private String nomeTime;
public Atleta() {

}
}
public class Time {
private String nomeAtleta;
private ArrayList<Atleta> atelta;
public Time() {

Atleta = new ArrayList<Atleta>();
}

public void add(Atleta umAtleta) {

Atleta.add(umAtleta);
```

Composição:

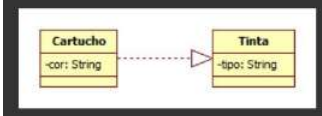
A **COMPOSIÇÃO** é uma agregação mais forte; nela, a existência do Objeto-Parte NÃO faz sentido se o Objeto-Todo não existir.




```
public Peca() { private int codigoPedido;
}

}
public class Pedido {
private int idPedido;
private ArrayList<ItemPedido> itemPedido;
public Pedido() {
itemPedido = new ArrayList<ItemPedido>();
}
public void add(ItemPedido umPedido) {
ItemPedido.add(umPedido);
}
}
```

Dependência:

A **DEPENDÊNCIA** deixa explícito que uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. Os objetos de uma classe esporadicamente usam serviços dos objetos de outra classe.




```
public class Cartucho {
public Cartucho (String cor) {
}

public Cartucho() {
}
}

public class Tinta {
public Tinta() {
}
public void escolhaTinta (int tipoTinta, Tinta qtde) {

}

}
```

Exceção (Exception):

Os erros em Java são, normalmente, chamados de exceptions. Uma exceção, como sugere o significado da palavra, é uma situação que normalmente não ocorre (ou não deveria ocorrer), algo estranho ou inesperado provocado no sistema.

Para tratar os erros podemos usar um bloco try-catch-finally.

É a principal estrutura para captura de erros em Java. O código tentará (try) executar o bloco de código que pode gerar uma exceção e, caso isso ocorra, o erro gerado será capturado pelo catch, que possui um parâmetro de exceção (identificação do erro) seguido por um bloco de código que o captura e, assim, permite o tratamento. É possível definir vários catch para cada try.

O finally é opcional e, se for usado, é colocado depois do último catch. Veja o Exemplo:

```
try
{
    for (int i = 0; i <= 15; i++)
    {
        cc.deposita(i + 1000);
        System.out.println(cc.getSaldo());
        if (i == 5)
        {
            cc = null;
        }
    }
} catch (NullPointerException e)
  {
      System.out.println("erro: " + e);
  }
  finally
  {

  }
```

```
inicio do main
inicio do metodo1
inicio do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
erro: java.lang.NullPointerException
fim do main
```