Capítulo 4

Lógica de programação aplicada à linguagem Java

- A Plataforma de desenvolvimento
- Origem da orientação a objetos
- UML (Unified Modeling Language)
- Eclipse
- Orientação a objetos (1)
- Entrada e saída de dados
- Assinatura de métodos
- Estruturas e recursos da linguagem Java
- Orientação a objetos (2)
- Padrões de desenvolvimento de sistemas
- Interfaces gráficas
- Tratamento de exceções
- Conexão com bancos de dados

Smalltalk-80, ou simplesmente Smalltalk, é uma linguagem de programação puramente orientada objetos. Isso significa que em Smalltalk tudo é objeto: os números, as classes, os métodos, os blocos de código etc. Tecnicamente, todo elemento é um objeto de primeira ordem.

Java é uma linguagem de programação desenvolvida nos anos 1990 por uma equipe de programadores chefiada por James Gosling, na Sun Microsystems, conglomerado norte-americano da área de informática (leia o quadro *Quem é quem no mundo Java*). Para conhecê-la mais profundamente, é preciso voltar um pouco no tempo e saber quais foram suas antecessoras: a BCPL e B, a C e a C++ (confira no quadro *A evolução da linguagem de programação*). Por trás de todo o sucesso do Java está um projeto de pesquisa corporativa batizado de Green, financiado em 1991 pela norte-americana Sun Microsystems. A Sun, cujo nome faz referência à Standford University, fabrica computadores, semicondutores e softwares. Tem sede no Silicon Valley (Vale do Silício), em Santa Clara, Califórnia (EUA) e subsidiárias em Hillsboro, no estado do Oregon (EUA), e em Linlithgow, na Escócia. Gosling e a equipe que tocou o projeto para a Sun apostavam na convergência dos computadores com outros equipamentos e eletrodomésticos e produziram um controle remoto com interface gráfica touchscreen (toque na tela) batizado de *7 (Star Seven) (figura 61). Para o desenvolvimento de parte do sistema operacional e dos aplicativos do *7, foi criada uma linguagem de programação baseada no C++ e no **smalltalk**, chamada de Oak (carvalho, em inglês), em homenagem a uma árvore que podia ser vista da janela da sede da Sun. Descobriu-se mais tarde que já havia uma linguagem de programação chamada Oak, o que implicava a necessidade de se buscar outro nome para a desenvolvida na empresa e assim nasceu o nome Java. Há, porém, outras versões para a origem do nome. Uma delas diz que foi escolhido quando a equipe visitou uma cafeteria local que tinha o nome Java, em referência à cidade de origem de um tipo de café importado pelo estabelecimento. Essa versão explica a xícara estilizada estampada no logo do Java (figura 62).

Figura 61
(à esquerda)
*7 StarSeven,
o ponto de partida.

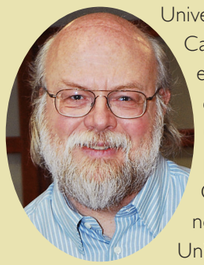
Figura 62
Logotipo do Java, com a xícara estilizada.

O projeto Green passou por algumas dificuldades. O mercado de dispositivos eletrônicos inteligentes, voltado para o consumo popular, no início da década de 1990, não se desenvolvia tão rápido como a Sun havia previsto. Por isso, corria o risco de ser cancelado. Por uma feliz coincidência, a www (World Wide Web, rede mundial

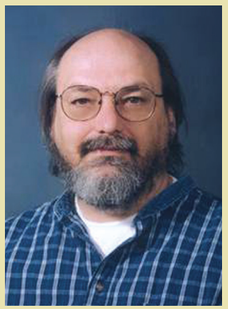
Quem é quem no mundo Java

James Gosling

Programador canadense conhecido como o pai da linguagem Java. Em 1977, tornou-se bacharel em Ciência da Computação pela Universidade de Calgary, no Canadá, e em 1983, PhD em Ciência da Computação pela Universidade Carnegie Mellon, nos Estados Unidos.



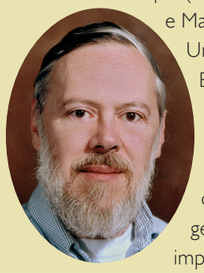
Kenneth Thompson



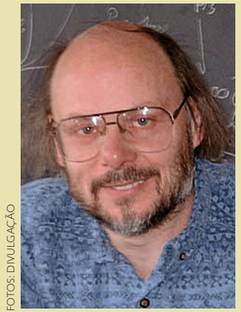
(ou Ken Thompson) - cientista computacional conhecido por sua influência no sistema operacional UNIX. Nasceu em 1943 em Nova Orleans, Louisiana, EUA, e tem mestrado pela UC Berkeley. Foi o criador da linguagem B, em 1969, no centro de pesquisas da AT&T Bell Labs, onde nasceram tecnologias consideradas revolucionárias (desde cabos de telefone, transistores, LEDs e lasers até linguagem de programação C e o sistema operativo Unix, que hoje é de propriedade do The Open Group, consórcio formado por empresas de informática).

Dennis MacAlistair Ritchie

É um cientista da computação notável por sua participação no desenvolvimento de linguagens e sistemas. Nascido em Bronxville, Nova Yorque (EUA), formou-se em Física e Matemática Aplicada pela Universidade de Harvard. Em 1983, Richie e Ken Thompson receberam o prêmio Turing pelo desenvolvimento da teoria de sistemas operacionais genéricos e, especialmente, pela implementação do sistema UNIX.



Bjarne Stroustrup



FOTOS: DIVULGAÇÃO

Cientista da computação dinamarquês e professor catedrático da Universidade do Texas, EUA. Conhecido como o pai da linguagem de programação C++.

de computadores) se popularizou em 1993 e a Sun viu o Java ser utilizado para adicionar conteúdo dinâmico às páginas da web, como interatividade e animações.

Além de salvar o projeto Green, a expansão da World Wide Web deu início ao desenvolvimento do Java como uma linguagem comercial. Essa mudança já trazia a ideia de estimular a interatividade entre os ambientes computacionais (mainframes, PCs etc.), os eletrodomésticos (que cada vez mais têm programação embarcada), equipamentos eletrônicos (que evoluiriam para os dispositivos móveis) e a então recém-nascida internet.

A Sun anunciou o Java formalmente durante uma conferência, em maio de 1995. A linguagem chamou a atenção da comunidade de negócios, devido ao enorme interesse que a web despertava na época. Além disso, também tinha o grande apelo da portabilidade: *Write once, Run anywhere*.

Por isso, o Java passou a ser utilizado para desenvolver aplicativos corporativos de grande porte e aprimorar a funcionalidade de servidores web (os computadores que fornecem o conteúdo http://www.sis.pitt.edu/~mbsclass/hall_of_fame/images/thompson.jpg que vemos em nossos navegadores da internet). Essa linguagem também passou a ser usada no desenvolvimento de aplicativos para dispositivos móveis (telefones celulares, pagers e PDAs, por exemplo).

4.1. A plataforma de desenvolvimento

O Java e as tecnologias envolvidas na orientação a objetos têm uma característica peculiar que provoca confusão e dúvidas entre a maioria das pessoas que começam a estudar o assunto. Há um universo de siglas e acrônimos que permeiam todas as fases do desenvolvimento e também formam uma verdadeira sopa de letrinhas. Mas vamos nos ater ao ambiente de desenvolvimento JSE. O Java começou na versão 1.0 (parece óbvio, mas nesse caso a sequência de versões não é tão previsível assim). Na versão 1.2, houve uma grande quantidade de atualizações e correções e um significativo aumento nas APIs (banco de recursos da linguagem, que será descrito mais adiante). Nessa etapa, a Sun resolveu trocar a nomenclatura de Java para Java2, para diminuir a confusão entre Java e Javascript e divulgar as melhorias trazidas pela nova versão. Mas que fique bem claro: não há versão do Java 2.0. O "2" foi incorporado ao nome (exemplo: Java2 1.2). Depois, vieram o Java2 1.3 e 1.4. No Java 1.5, a Sun alterou novamente o nome para Java 5. Até a versão 1.4, existia uma terceira numeração (1.3.1, 1.4.1, 1.4.2 etc.), indicando correções de erros e melhorias. A partir do Java 5, começaram a surgir diversas atualizações, como Java 5 update 7, e Java6 1.6 update 16.

4.1.1. JRE (Java Runtime Environment)

Os ambientes de desenvolvimento Java contêm um pacote (conjunto de softwares que se complementam e interagem para determinada finalidade) chamado JRE (Java Runtime Environment). Ele é executado como um aplicativo do sistema operacional e interpreta a execução de programas Java de forma performática, segura e em escalas. A JRE é composta pela JVM (Java Virtual Machine, ou máquina virtual Java), por um conjunto de API's (Application Programming Interface ou interface de programação de aplicações) e por alguns pacotes de desenvolvimento.

4.1.1.1. JVM (Java Virtual Machine)

A máquina virtual Java, tradução literal da sigla em inglês JVM, é um programa que carrega e executa os programas desenvolvidos em Java, convertendo os bytecodes (código fonte pré-compilado) em código executável de máquina.

A evolução da linguagem de programação

BCPL

Acrônimo para Basic Combined Programming Language (Linguagem de Programação Básica Combinada), a BCPL é uma linguagem de programação voltada para o desenvolvimento de softwares para sistemas operacionais e compiladores. Foi criada por Martin Richards em 1966 e utilizada, anos depois, por Ken Thompson para desenvolver a B e, depois, a C.

B

Sucessora da BCPL, a B foi desenvolvida no famoso centro de pesquisas da AT&T Bell Labs em 1969 – um trabalho capitaneado por Ken Thompson com contribuições de Dennis Ritchie. Ken Thompson utilizou a linguagem B para criar as primeiras versões do sistema operacional UNIX. Hoje é obsoleta, mas seu valor é incontestável por ter sido a precursora da C. Por isso, é uma das mais populares no mundo.

C

É uma linguagem de programação compilada, estruturada, imperativa, processual, de alto nível e padronizada. Foi criada em 1972, por Dennis Ritchie, também no AT&T Bell Labs, como base para o desenvolvimento do sistema operacional UNIX (escrito em Assembly originalmente). A maior parte do código empregado na criação dos sistemas operacionais de equipamentos, como desktops, laptops, estações de trabalho e pequenos servidores, é escrita em C ou C+ +.

C++

De alto nível, com facilidades para o uso em baixo nível, é uma liguagem comercial multiparadigma e de uso geral. Muito popular, desde os anos 1990, também está entre as preferidas nos meios universitários em virtude de seu grande desempenho e enorme base de usuários. Foi desenvolvida por Bjarne Stroustrup (primeiramente, com o nome C with Classes, que significa C com classes em português), em 1983, no Bell Labs, como um elemento adicional à linguagem C. A linguagem C++ oferece vários recursos que complementam e aprimoram a linguagem C. Sua principal característica, no entanto, está na adoção da programação orientada a objetos, técnica de programação que é abordada em todo este capítulo.

A JVM é responsável pelo gerenciamento dos programas Java, à medida que são executados, simulam uma CPU e a memória (daí o nome máquina virtual). Graças à JVM, os programas escritos em Java podem funcionar em qualquer plataforma de hardware e software que possua uma versão da JVM. Ou seja, cada sistema operacional compatível com o Java possui sua própria versão de JVM, tornando os programas desenvolvidos em Java independentes em relação à plataforma (hardware e software) onde estão sendo executados. Configura-se, assim, uma das principais características da linguagem Java: a portabilidade.

4.1.1.2. API (Application Programming Interface)

A interface de programação de aplicação, tradução literal da sigla em inglês API, é um conjunto de recursos (classes e interfaces) com funcionalidades padrão, já implementados, e que possuem ampla documentação. Assim, os programadores podem utilizar esses recursos sem precisar saber como foram desenvolvidos.

4.1.2. Ambientes de desenvolvimento

4.1.2.1. JSE (Java Standard Edition)

Em português, é a edição Java padrão. Contém todo o ambiente necessário para a criação e execução de aplicações Java, incluindo a máquina virtual Java (JVM), o compilador Java (JRE), um conjunto de APIs e outras ferramentas utilitárias. É voltada para o desenvolvimento de aplicações desktop e web de pequeno porte, executadas em PCs e em servidores. O JSE é considerado o ambiente principal, já que serviu de base para o desenvolvimento do JEE (Java Enterprise Edition ou edição Java corporativa) e do JME (Java Micro Edition ou edição Java micro, voltada para microdispositivos). Por ser a plataforma mais abrangente do Java, a JSE é a mais indicada para quem quer aprender a linguagem.

4.1.2.2. JEE (Java Enterprise Edition)

Servidores de aplicação são programas que fornecem uma série de recursos para suporte de aplicações corporativas, na web, de forma segura e consistente.

A plataforma JEE (sigla em inglês, que literalmente significa edição empresarial Java) contém bibliotecas que oferecem facilidades para a implementação de softwares Java distribuídos, tolerantes a falhas. Voltada para aplicações multicamadas baseadas em componentes que são executados em um **servidor de aplicações**, a JEE é considerada, mais do que uma linguagem, um padrão de desenvolvimento. Nela, o desenvolvedor de software segue obrigatoriamente determinadas regras para que seu sistema possa se comunicar com outros de maneira automática. Destina-se a aplicações corporativas de grande porte e que usam frequentemente o EJB (Enterprise Java Beans, um padrão com estrutura de aplicações distribuídas) e servidores de aplicação. Outro recurso desse ambiente são as aplicações que rodam no lado servidor em uma arquitetura cliente/servidor, conhecidas como servlets.

4.1.2.3. JME (Java Micro Edition)

A plataforma Java Micro Edition (literalmente, edição Java micro, voltada para microdispositivos) permite o desenvolvimento de softwares para sistemas e aplicações embarcadas, ou seja, aquelas que rodam em um dispositivo específico. É bastante indicada para aparelhos compactos, como telefone celular, PDA (Personal Digital Assistant ou assistente digital pessoal), controles remotos etc. Contém configurações e bibliotecas desenvolvidas especialmente para os portáteis. Com esse instrumento, o desenvolvedor tem mais facilidade para lidar com as limitações de processamento e memória do equipamento.

A plataforma Java Micro Editon é indicada para aparelhos como o Personal Digital Assistant.

4.1.2.4. JDK (Java Development Kit)

A JDK (ou JSDK Java Software Development Kit, kit de desenvolvimento de programas Java) é o ambiente completo de desenvolvimento Java. Existe um JDK para cada ambiente (JSE, JEE e JME). Esse kit deve ser instalado na máquina para possibilitar a criação de aplicações Java.

4.1.2.5. JSE: nosso foco

O JSE é o mais indicado para os programadores iniciantes (ou mesmo veteranos de outras linguagens) e será o foco de todos os exemplos deste capítulo. Mas, apesar dos diferentes ambientes de desenvolvimento, trata-se da mesma linguagem de programação. Portanto, a maioria dos recursos que serão abordados é comum e perfeitamente aplicável a outros ambientes, principalmente no que diz respeito à orientação a objetos.

4.1.3. A portabilidade

Em primeiro lugar, é preciso entender o que é um programa compilado e um programa interpretado. Compilação é o processo de conversão do código fonte (texto escrito segundo as regras de determinada linguagem de programação) para a linguagem de máquina (programa pronto para ser executado). Nesse processo, são feitas várias referências (e consequentemente suas dependências) em

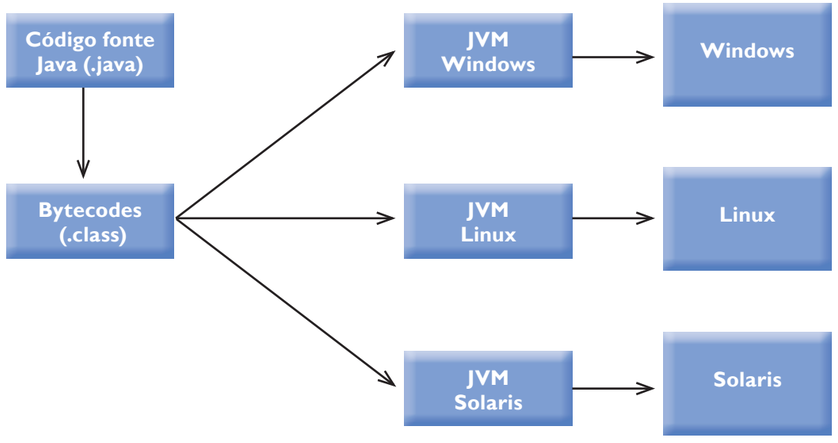


Figura 63
Portabilidade Java.

relação ao sistema operacional no qual o programa é gerado. Já em um programa interpretado a execução ocorre por meio da leitura, interpretação e execução direta do código fonte (não existe o processo de compilação).

Na comparação entre esses dois modelos de desenvolvimento, é possível perceber que os programas compilados oferecem melhor desempenho (velocidade de execução), enquanto os interpretados são bem mais lentos. Em contrapartida, os compilados são totalmente dependentes da plataforma e do sistema operacional em que foram gerados, ao contrário dos interpretados.

O Java se aproveita do que há de bom nos dois modelos e descarta os pontos negativos de cada um, por meio da JVM. Um código fonte Java é pré-compilado e convertido em bytecode pela JRE (e não pelo sistema operacional). O **bytecode** é gerado por meio de recursos da JRE e executado sobre a JVM, ou seja, sem interação direta com o sistema operacional da máquina em que foi desenvolvido. No entanto, a JVM precisa interagir com o sistema operacional e, por isso, tem versões para diferentes sistemas operacionais.

Código fonte Java é um arquivo de texto escrito de acordo com as regras da linguagem Java e salvo com a extensão .java. Já a expressão bytecode se refere a um arquivo convertido em linguagem de máquina e salvo com o mesmo nome do código fonte, porém com a extensão.class.

Como tudo passa pela JVM, essa plataforma pode tirar métricas, decidir qual é o melhor lugar para alocar ou suprimir memória, entre outras funções que antes ficavam sob a responsabilidade do sistema operacional. A figura 63 mostra como a portabilidade em aplicações Java é alcançada e qual o papel da JVM nesse contexto.

4.2. Origem da 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.).

4.2.1. 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.

4.3. UML (Unified Modeling Language)

A UML, sigla em inglês de linguagem modular unificada, é dedicada à especificação, visualização, construção e documentação que usa notação gráfica para

modelar softwares. Muito utilizada em empresas que desenvolvem aplicações orientadas a objetos, também está presente em várias publicações relacionadas com a linguagem Java (além de ser um instrumento interessante para elaboração de exercícios em aula). Por isso, escolhemos um de seus diagramas (de classes) para ilustrar nossos exemplos. É bom frisar, no entanto, que os recursos da UML vão muito além desses diagramas.

4.4. Eclipse

IDE (Integrated Development Environment, ou ambiente integrado de desenvolvimento) é um programa de computador que reúne recursos e ferramentas de apoio para agilizar o desenvolvimento de softwares. Aqui, trataremos da IDE Eclipse Classic 3.5.0, comumente conhecida por Galileo (as versões mais recentes ficam disponíveis para download no site htpp://www.eclipse.org/downloads/.)

A Eclipse Foundation é uma organização sem fins lucrativos que hospeda os projetos Eclipse e ajuda a manter tanto uma comunidade de openSource (grupo que desenvolve softwares com código de fonte aberto) como um conjunto de produtos e serviços complementares. O projeto Eclipse foi originalmente criado pela IBM em novembro de 2001 e apoiado por um consórcio de fornecedores de software. (Para mais informações sobre o Eclipse, acesse: htpp://www.eclipse.org).

O Eclipse é mantido pela **Eclipse Foundation**, que possui projetos que se concentram na criação de uma plataforma de desenvolvimento composta por quadros extensíveis e ferramentas para construção, implantação e gerenciamento de softwares em todo o seu ciclo de vida.

4.5. Orientação a objetos (I)

Uma vez familiarizados com uma parte do universo das siglas e com os ambientes de desenvolvimento relacionados à linguagem Java, é o momento de aprofundar o conhecimento do tema e das boas práticas de programação, na construção de softwares orientados a objetos. Para isso, nosso foco de estudo será a construção de um projeto. Para começar, é importante entender diversos conceitos que ajudarão a estruturar e a padronizar os programas desenvolvidos com essa técnica. A construção de softwares independentes, de fácil expansão, de alto nível de reusabilidade (a possibilidade de reutilizar trechos de um programa) e que viabilizam a interação entre programas e plataformas diferentes, depende dessa padronização, graças à adoção dos conceitos da orientação a objetos. Deixar de usar esses conceitos não gera, necessariamente, erros de compilação, de lógica ou de execução. Porém, faz com que os softwares gerados não apresentem as vantagens inerentes à orientação a objetos, como, justamente, o aumento da reusabilidade e a facilidade de manutenção e de expansão.

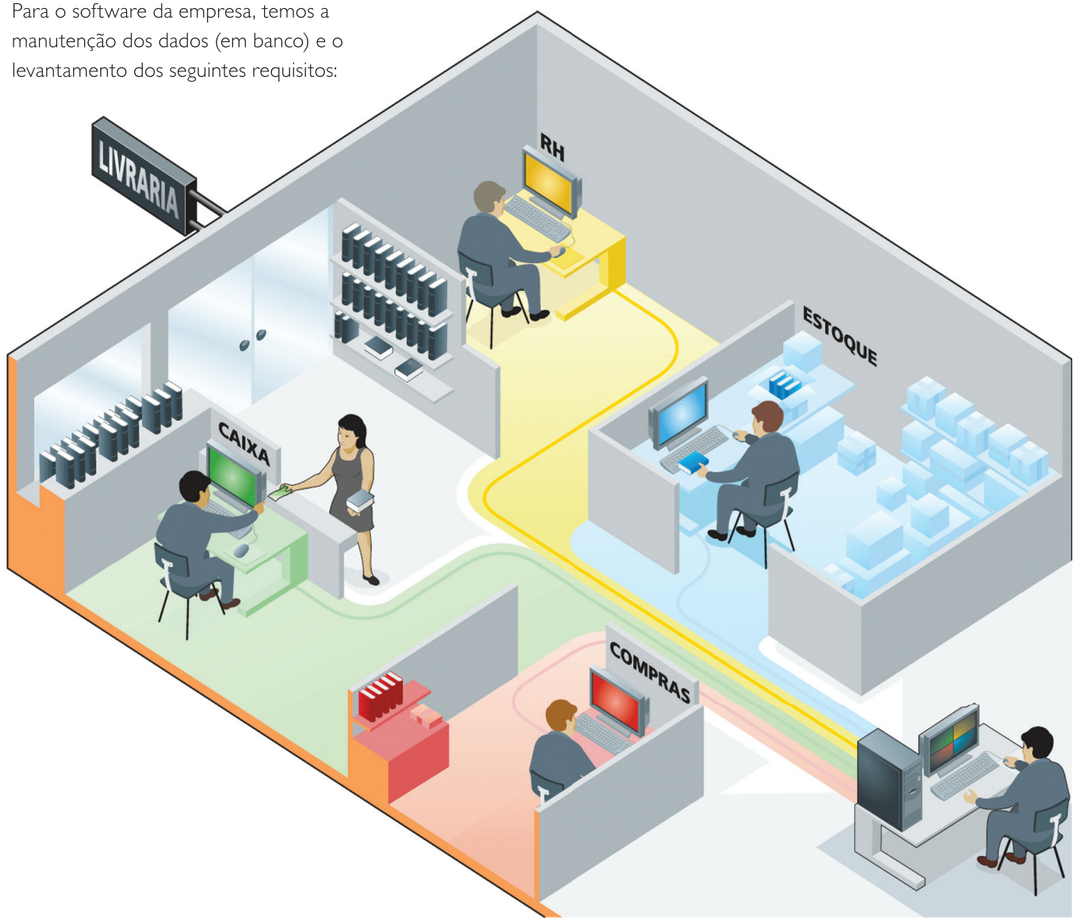
Outro termo comumente usado é convenção (de nomenclatura). Há alguns anos, cada programador escrevia seus códigos de uma maneira bem particular, criando seus próprios estilos e manias. Isso não atrapalhava a execução dos programas (desde que fosse utilizada uma boa lógica, além de adequados recursos das linguagens). Porém, com o passar dos anos e a evolução dos sistemas, o trabalho em equipe se tornou inevitável e vários programadores (muitas vezes distantes entre si, fisicamente), passaram a dar manutenção em códigos comuns. A partir daí, surgiu a necessidade de adotar determinados padrões de escrita de códigos. Há algumas convenções de nomenclatura bem antigas e consolidadas, como a notação húngara, amplamente utilizada em C e C++. Mas, na prática, as linguagens e até mesmo as empresas acabam criando suas próprias convenções.

No caso do Java, existe uma documentação específica (disponível no site http://www.java.sun.com/docs/codeconv/) que define sua própria convenção de nomenclatura, cuja utilização será destacada durante a elaboração dos códigos fonte. Tal como ocorre com os conceitos, deixar de usá-la não gera erros de lógica, de compilação ou de execução. Seguiremos, entretanto, à risca as recomendações de conceitos e convenções, para que sejam conhecidas em profundidade e também em nome da implantação de boas práticas de programação.

O projeto da Livraria Duke

A partir desse ponto, vamos desenvolver um controle de estoque e movimentação para uma livraria fictícia, chamada Duke. Todos os exemplos criados a seguir serão reaproveitados para a construção desse projeto.
Para o software da empresa, temos a manutenção dos dados (em banco) e o levantamento dos seguintes requisitos:

- **Controle de clientes:** vendas.
- **Controle de funcionários:** cálculo de salários.
- **Controle de produtos:** compras, vendas e estoques disponíveis.
- **Controle de fornecedores:** compras realizadas de cada fornecedor.
- **Controle de compra e venda:** o sistema manterá um registro de todas as movimentações realizadas de acordo com as regras da livraria.



4.5.1. Componentes elementares

4.5.1.1. Classes

Em Java, tudo é definido e organizado em classes (a própria linguagem é estruturada dessa forma). Inclusive, existem autores que defendem ser correto chamar a orientação a objeto de orientação a classes. As classes exercem diversos papéis em programas Java, a princípio vamos nos ater a duas abordagens:

- **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). Poderia ter qualquer nome, desde que respeitadas as regras de nomenclatura de classes. Será chamada, então, de "Principal", como referência ao método main.

Nossa primeira classe será a representação de uma pessoa segundo os critérios da orientação a objetos. Para isso, utilizaremos um diagrama de classe (UML), um critério que valerá para os demais exemplos desse capítulo (figura 64).

Do projeto ao automóvel, assim como da classe ao objeto.

EQUIPE EESC-USP BAJA SAE

As classes de modelagem podem ser comparadas a moldes ou formas que definem as características e os comportamentos dos objetos criados a partir delas. Vale traçar um paralelo com o projeto de um automóvel. Os engenheiros definem as medidas, a quantidade de portas, a potência do motor, a capacidade em relação ao número de passageiros, a localização do estepe, dentre outras descrições necessárias para a fabricação de um veículo.

Durante a elaboração do projeto, é óbvio que o automóvel ainda não existe. Porém, quando for fabricado, todas as especificações previamente definidas no desenho terão de ser seguidas à risca. Os diferentes modelos desse mesmo veículo terão detalhes diferenciais, como cor da lataria, tipo das rodas, material do banco, acessórios etc. Nada, entretanto, que altere as características originais, como um todo. Serão informações acrescentadas aos elementos já definidos no projeto (atributos).

O funcionamento básico do automóvel também foi definido no projeto. O motorista poderá acelerar, frear, trocar marchas, manobrar etc. Essas ações, assim como a respectiva resposta do carro (métodos), permitem uma outra analogia com nosso assunto. O motorista (que seria no nosso caso o programador) não precisa ter conhecimentos profundos sobre mecânica para, por exemplo, acelerar o automóvel. Internamente, o veículo possui um complexo mecanismo que aciona as partes responsáveis pela injeção e pela combustão. Esta, por sua vez, gera uma pressão sobre outras engrenagens do motor, impulsionando as rodas e fazendo o carro se deslocar – processos também especificados no projeto. Para o motorista, quando o objetivo é velocidade, basta saber que é preciso pisar no pedal do acelerador e que quanto mais pressão for feita mais rápido andará o veículo. Resumindo: para uso diário do automóvel (objeto) não é preciso conhecer detalhes de como o funcionamento dos mecanismos foi definido no projeto (classe). Basta operá-los (chamar os métodos, obedecendo à sua assinatura – como veremos, em seguida). Da mesma forma, o programador não precisa saber, em detalhes, como a classe System, por exemplo, foi desenvolvida, mas sim saber como utilizá-la para apresentar uma informação na tela.

4.5.1.1.1. Atributos

As informações (dados) que definem as características e os valores pertinentes à classe e que serão armazenados nos (futuros) objetos são chamadas de atributos. Também são conhecidos como variáveis de instância. Para não confundir com variáveis comuns, vamos chamá-los sempre de atributos.

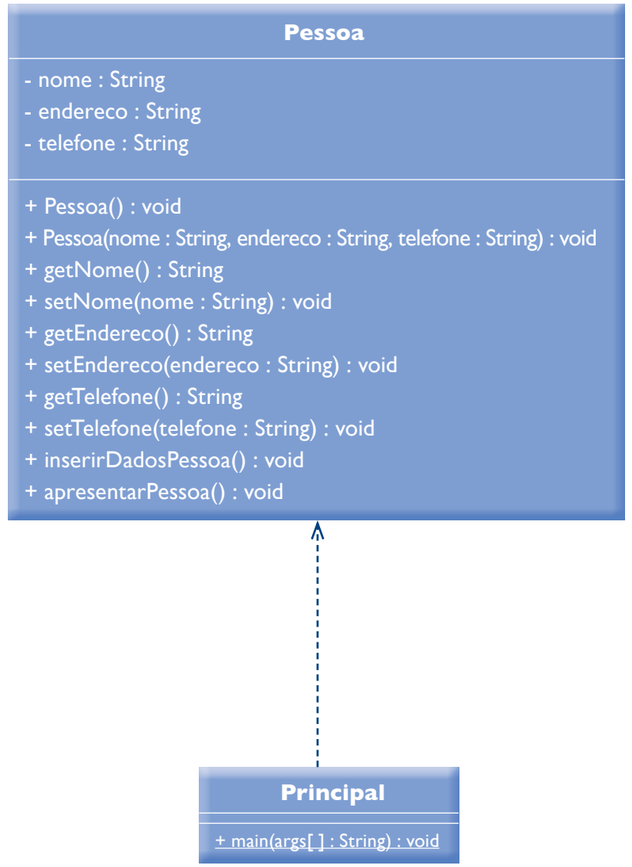
4.5.1.1.2. Métodos

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.

4.5.1.1.3. Detalhando o diagrama de classe

Na descrição das informações contidas no diagrama de classe, novos termos aparecerão. Nesse momento, porém, o importante é perceber que o diagra-

Figura 64
Diagrama de classe (UML) da classe Pessoa.



ma traz detalhes dos atributos e métodos que compõem a classe. A tabela 11 descreve os detalhes contidos nos três compartimentos (nome, atributos e métodos) do diagrama de classe.

4.5.1.1.4. Codificação da classe Pessoa

A abertura de uma classe é definida com seu modificador de acesso (detalhado a seguir), que, por sua vez, dá a visibilidade (nesse caso, pública), além do comando class e do nome da classe. A definição da classe fica delimitada entre as chaves { }, que estabelecem o início e o fim de um bloco de código – da linha 1 a 57. Observe, na figura 65, a codificação da classe Pessoa. Perceba que parte do código foi suprimida, mais precisamente a codificação dos métodos (ficando visível somente a assinatura – ou abertura – dos métodos), os quais serão detalhados mais adiante. É possível perceber as partes ocultas no código por meio de um sinal de + na frente do número da linha.

Nome da classe **Pessoa**	• **Por convenção, todo nome de classe começa com letra maiúscula.**
Atributos - nome : String - endereco : String - telefone : String	• **O sinal de menos "-" indica a visibilidade do atributo. Nesse caso, privado (private em inglês).** • **Por convenção, todos os nomes de atributos são escritos com letras minúsculas.** • **Depois é especificado o tipo de dado que será armazenado no atributo**
Métodos + Pessoa() : void + Pessoa(nome : String, endereco : String, telefone : String) : void + getNome() : String + setNome(nome : String) : void + getEndereco() : String + setEndereco(endereco : String) : void + getTelefone() : String + setTelefone(telefone : String) : void + inserirDadosPessoa() : void + apresentarPessoa() : void	• **O sinal de mais "+" indica a visibilidade do método, nesse caso, público (public em inglês).** • **Por convenção, o nome dos métodos é escrito com letras minúsculas.** • **Os únicos métodos que têm exatamente o mesmo nome da classe são os construtores. Também iniciam com letras maiúsculas.** • **Nomes compostos por várias palavras começam com letra minúscula e têm a primeira letra das demais palavras maiúscula, sem espaço, traço, underline, ou qualquer outro separador. Por exemplo, "apresentar Pessoa".** • **Todo método tem parênteses na frente do nome. Servem para definir os seus parâmetros. Um método pode ou não ter parâmetros.** • **Por último, a definição do retorno. Se o método tiver retorno, é especificado o tipo, por exemplo, String e se não tiver é colocada a palavra void (vazio).**

Tabela 11 Detalhes dos três compartimentos do diagrama de classe.

Figura 65
Codificação da classe Pessoa.

```
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 }
```

Quando analisamos a codificação da classe, percebemos que ali está tudo o que foi definido no diagrama de classe. Depois da abertura da classe, vem a declaração dos atributos (da linha 4 a 6), composta pelo modificador de acesso (private), pela definição do tipo de dado a ser armazenado (String) e pelo nome do atributo (por exemplo nome). Em comparação com o diagrama de bloco, a sequência muda (no diagrama vem o modificador de acesso, o nome e o tipo conforme o padrão da UML). Mas as informações são as mesmas.

Como definição de um comportamento específico da classe, temos o método apresentarPessoa, que mostrará o conteúdo dos atributos em prompt (simulado no eclipse pelo painel console). Conforme definido no diagrama, o método é público (public), não retorna valor (void), possui o nome apresentarPessoa e não recebe pa-

Figura 66
Codificação 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     }
```


râmetros "( )". Essas especificações, mais precisamente o nome e os parâmetros (a quantidade e o tipo), definem a assinatura do método. Isso será detalhado depois, assim como as linhas de código que fazem a apresentação dos dados. No trecho ilustrado pela figura 66, observamos a codificação do método apresentarPessoa. Perceba que toda a linha de instrução termina com ponto-e-vírgula.

4.5.1.1.5. Comentários

Existem algumas formas de inserir comentários no código, como ilustra a figura 67. Essa prática é aconselhável para a futura geração de documentação do sistema e também possibilita o esclarecimento do código em pontos específicos. Isso pode ser muito útil para você e outros programadores que trabalhem no mesmo código.

Figura 67
Inserção de comentários no código.

```
// Comentário em uma linha


// Comentário
// em várias
// linhas


/* Comentário
 * em várias
 * linhas
 */


/* Comentário
   em várias
   linhas
*/


/** Comentário no formato reconhecido
 *  por um utilitário de documentação
 *  chamado javadoc fornecido pela
 *  SUN junto com o JDK
 */
```

4.5.1.2. Objetos

A partir do momento em que a classe de modelagem está completa, é possível criar um objeto e manipulá-lo. Outra vez, surgirão termos novos, explicados mais adiante. O importante, agora, é perceber que essa é a classe Principal (salva em um arquivo separado da classe Pessoa). Nela está definido o método main (principal), dentro do qual estão as linhas de código que criam e acessam o objeto.

Todo projeto Java possui, obrigatoriamente, um método main. É esse que a JRE procura para ser executado primeiro. A partir dele, o restante do

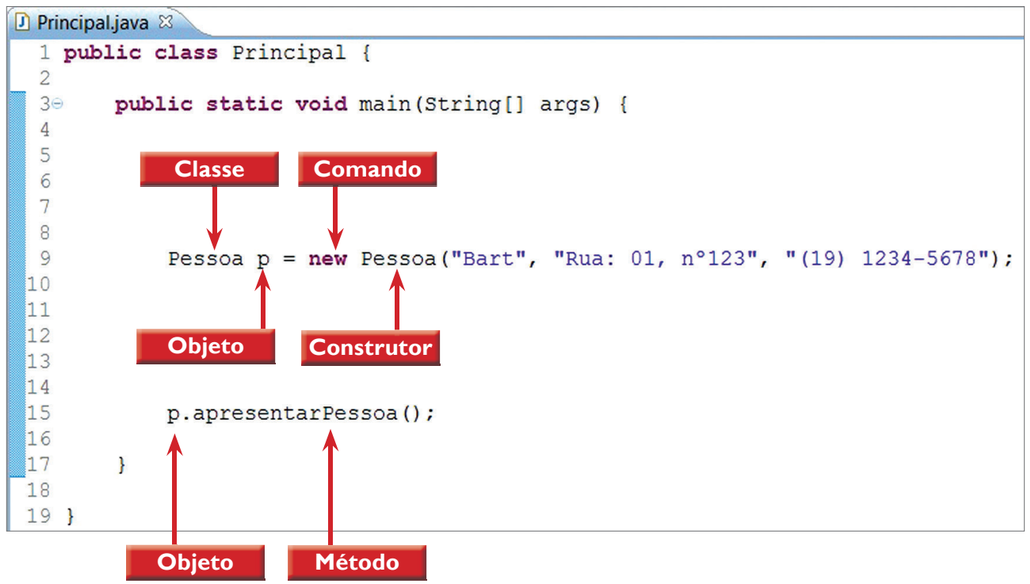


Figura 68
Codificação da classe Principal.

sistema é acessado e executado. A assinatura do método main será sempre feita dessa forma.

Observe a figura 68. Vamos à linha 09, na qual é realizada a criação (instanciação) do objeto "p" do tipo Pessoa. Veja como isso ocorre:

- Primeiro houve a abstração da entidade do mundo real a ser representada em nosso programa (no caso, uma pessoa). Isso aconteceu na definição das informações que deveríamos armazenar (atributos) e nas ações que deveríamos programar (métodos).

- Depois, desenhamos essa entidade (Pessoa), utilizando um diagrama de classe (pertencente à especificação da UML) e definimos a classe.

- Ao definirmos a classe, detalhamos todos os atributos (dados) e métodos (ações) pertencentes a essa classe.

- Uma vez definida a classe (o novo tipo), podemos criar (instanciar) um objeto para ser utilizado. Lembre-se: na classe são feitas as especificações do objeto, assim como o projeto do automóvel. Nós não manipulamos a classe, da mesma forma que não dirigimos o projeto do automóvel. Devemos instanciar (criar um objeto) a partir da classe criada, assim como o carro deve ser fabricado a partir do projeto. Daí, poderemos manipular as informações sobre o cliente dentro do programa, o que equivale a dirigir o carro no mundo real.

- Existe a possibilidade de acessar atributos e métodos de uma classe por meio de outra, sem a necessidade de instanciar um objeto. Isso pode ser feito com ajuda do modificador static (estático), na classe Math do Java, por exemplo.

- Para a criação do objeto na linha 09, primeiramente, é informado o tipo (Pessoa), depois o nome do novo objeto (nesse caso, "p"), = new (novo - comando que cria uma instância da classe que chamamos de objeto). Depois, é utilizado um método construtor (o único tipo de método que tem o mesmo nome da classe, inclusive com letra maiúscula), que permite a inserção de dados (informações) nos atributos já no momento de sua criação. O objeto p do tipo Pessoa foi criado e contém os valores "Bart" no atributo nome, "Rua: 01, nº 123" no atributo endereço e "(19) 1234-5678" no atributo telefone.

- Em seguida, com o objeto já criado, seu método apresentarPessoa será chamado e as mensagens, apresentadas conforme a figura 69.

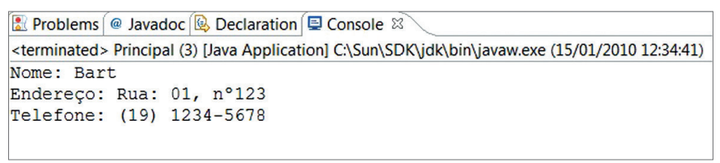


Figura 69
Saída do método apresentarPessoa().

4.5.2. 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 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.

4.5.2.1. 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).

Porém, a partir de uma instância da classe Pessoa, isto é, de um objeto do tipo Pessoa, o atributo fica inacessível.

4.5.2.2. 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.

4.5.2.3. 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).

O uso de packages será detalhado mais adiante.

4.5.3. 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 70.

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

Figura 70
Codificação dos construtores na classe Pessoa.

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

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


aos 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 mesmo acontece no método this(), no qual os valores entre parênteses referem-se aos atributos na mesma sequência.

4.5.4. Garbage Colletor (Coletor de lixo)

Em um sistema orientado a objetos, vários deles são **instanciados** (criados durante sua execução). Para que se possa utilizar esse objeto, é criada uma referência a esse endereço de memória onde ele está armazenado. Toda vez que fazemos uma referência a um objeto, a JVM (mais precisamente um recurso dela) registra essa informação em uma espécie de contador. Assim, é feito o controle de quantas referências estão "apontando" para aquele espaço da memória (objeto). Quando o contador está em zero, ou seja, não há referência, o objeto se torna um candidato a ser removido da memória.

Instanciar um objeto é alocar (reservar) um espaço em memória (RAM - Random Access Memory ou memória de acesso randômico).

Periodicamente, em intervalos calculados pela JVM, esses objetos sem referências são retirados da memória e o espaço alocado volta a ficar disponível, dando lugar a novos. O recurso responsável por essa limpeza da memória é chamado de Garbage Collector (coletor de lixo). Tem como responsabilidade rastrear a memória, identificar quantas referências existem para cada objeto e executar um processo que elimine os objetos sem referências. Substitui, então, os métodos destrutores e o gerenciamento de ponteiros (referências a espaços de memória) existentes em outras linguagens.

4.5.5. 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). Adotaremos o uso do this em todos os nossos exemplos.

4.5.6. Encapsulamento

No exemplo da classe Pessoa, utilizamos mais um conceito da programação orientada a objetos. Quando definimos que um atributo (ou método) tem a visibilidade privada (modificador de acesso private), limitamos o acesso somente à sua classe. Podemos imaginar que, depois de instanciado o objeto, esse atributo fica protegido (encapsulado) dentro desse objeto. Assim, não é possível acessá-lo de nenhuma outra classe ou objeto. Tal característica é extremamente importante para proteger atributos que não devem ter seus conteúdos vulneráveis a acessos indevidos.

Imagine que o valor máximo que um cliente pode comprar em uma loja está definido em um atributo limiteDeCredito em uma classe Cliente (que dá ori-

gem aos clientes do sistema). Seja por descuido ou por má fé, se o valor desse atributo for alterado para um valor exorbitante, ocorrerá a seguinte situação no sistema: o cliente (do mundo real) ficará liberado para comprar até esse valor exorbitante (tendo ou não condições para pagar). Normalmente, os empreendimentos comerciais definem limites de crédito para seus clientes, tomando por base a renda mensal de cada um, o tempo de cadastro, o histórico de pagamentos etc. Enfim, há uma regra para que esse limite de crédito seja calculado e/ou atualizado.

Agora, imagine outra situação. Determinada classe do sistema é responsável pelo processo de venda dessa mesma loja. Em algum momento, será necessário verificar se a quantidade de estoque disponível é suficiente para atender aos pedidos. Portanto, essa classe deve ter acesso ao atributo estoqueDisponível do produto que está sendo vendido. Tal acesso é necessário e autorizado pelo sistema. Porém, se o atributo estoqueDisponivel foi definido como privado, esse acesso não será possível. A solução para o problema é definir métodos de acesso para os atributos.

4.5.6.1. 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.

4.5.6.1.1. 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 (figura 71a).


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

Figura 71a
O método get.

4.5.6.1.2. Método set

Já os métodos responsáveis por escrever (alterar) o conteúdo de um atributo começam com a palavra set, seguida pelo nome desse atributo (figura 71b) .

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

Figura 71b
O método set.

4.5.7. Representação do encapsulamento em um objeto do tipo Pessoa

A figura 72 ilustra o encapsulamento em um objeto do tipo Pessoa. Imagine o objeto como uma esfera, que possui na parte interna (azul) os atributos (privados) que não têm acesso ao mundo externo (restante da aplicação). Já a camada externa (amarela) tem acesso ao mundo externo (pelo lado de fora) e à parte interna (pelo lado de dentro)

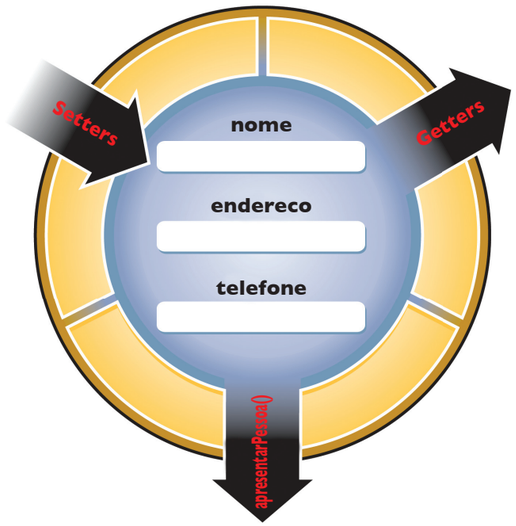


Figura 72
Representação gráfica do encapsulamento.

e funciona como uma película protetora do objeto. Tem passagens padrão que transferem informações de fora para dentro (setters) e de dentro para fora (getters). Além disso, têm algumas passagens (métodos) com finalidades específicas e que também podem entrar e/ou sair com informações. Esse tráfego acontece de acordo com regras de entrada e saída de informações definidas nas passagens (métodos). Em nossos exemplos, sempre serão utilizados getters e setters para acessar os atributos, garantindo os benefícios do encapsulamento.

4.5.8. Visão geral da classe Pessoa e sua estrutura

A figura 73 reproduz a codificação completa da classe Pessoa, destacando as camadas vistas até agora.

Figura 73
Codificação completa da classe Pessoa.

Pessoa.java

```
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 }
```

atributos

construtores

getters e setters (métodos de acesso)

método específico da classe Pessoa

4.6. Entrada e saída de dados

Já vimos como inserir dados nos atributos por meio do construtor e também como mostrar esses dados por meio do comando System.out.println(). Existem ainda diferentes formas de leitura e de escrita de dados em Java. Agora, para desenvolver exemplos mais elaborados, é preciso saber como realizar a entrada e a saída de dados utilizando a classe JOptionPane, localizada na API Swing=, que contém vários métodos com essa finalidade.

4.6.1. Declaração import

Os comandos utilizados nas classes devem estar acessíveis para que a JRE consiga interpretá-los e executá-los. Desse modo, um conjunto de comandos básicos fica disponível por padrão, mas a maioria dos recursos da linguagem necessita de uma indicação para a JRE localizá-los. Isso é feito pela declaração import, que deve ser inserida no começo da classe (antes da abertura) e é composta pelo caminho a partir da API até o local do recurso desejado. No caso da classe JOptionPane o import ficará assim como aparece na figura 74.

O uso do caractere especial * (asterisco) pode substituir parte do caminho especificado, como sugere o exemplo da figura 75.

Nesse caso, a interpretação é "posso utilizar qualquer recurso existente dentro da API Swing que pertence a API javax". A API Swing e seus recursos serão abordados com detalhes, quando conhecermos os componentes gráficos disponíveis no Java.

Figura 74
O import, na classe JOptionPane.

Pessoa.java

```
1 import javax.swing.JOptionPane;
```

Figura 75
O uso do caractere especial *.

Pessoa.java

```
1 import javax.swing.*;
```

4.6.2. Apresentação de dados (saída)

Para apresentação de mensagens, será usado o método showMessageDialog, pertencente à API Swing. Vamos alterar o método apresentarPessoa da classe Pessoa criada anteriormente, deixando da forma como aparece na figura 76.

Usar o método showMessageDialog requer a informação prévia de qual classe ele pertence (JOptionPane) e recebe, no mínimo, dois parâmetros. O primeiro é um componente do tipo object que, por ora, usaremos "null". Já o segundo é a mensagem propriamente dita, que pode ser composta por trechos de texto literal, com conteúdo de variáveis, ou retorno de métodos,

Figura 76
Método showMessageDialog.

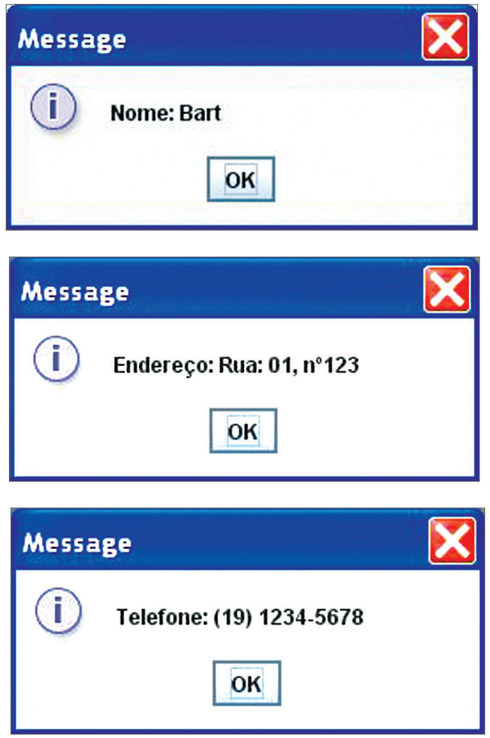
```
public void apresentarPessoa(){
    JOptionPane.showMessageDialog(null, "Nome: " + this.getNome());
    JOptionPane.showMessageDialog(null, "Endereço: " + this.getEndereco());
    JOptionPane.showMessageDialog(null, "Telefone: " + this.getTelefone());
}
```

como no exemplo da figura 76, no qual a mensagem gerada pela primeira linha será o texto "Nome:", juntando (somando as Strings) com o retorno do método getNome(), ou seja, o conteúdo do atributo nome. A execução do método apresentarPessoa ficará como ilustra a sequência da figura 77.

É possível realizar a apresentação da mensagem em uma única janela, concatenando todas as informações em uma única String (figura 78).

O sinal de mais (+) é o responsável pela concatenação (soma) dos trechos de textos e retorno de valor dos métodos get. O parâmetro "\n" faz uma quebra de linha na exibição da mensagem. Vale lembrar que quebrar a linha no editor (pressionando enter) não faz nenhuma diferença para o compilador; é interessante apenas para melhorar a visibilidade do código para o programador. A visualização do método apresentarPessoa depois da alteração ficará como ilustra a figura 79.

Figura 77
Execução do método apresentarPessoa.




```
public void apresentarPessoa(){
    JOptionPane.showMessageDialog(null, "Nome: " + this.getNome() +
                                        "\nEndereço: " + this.getEndereco() +
                                        "\nTelefone: " + this.getTelefone());
}
```

Figura 78
Alteração no método apresentarPessoa.

Figura 79
Apresentação da mensagem em uma única janela.

4.6.3. Leitura de dados (entrada)

Para leitura (entrada) de dados, será usado o método showInputDialog(). E, para testar, acrescentaremos o método inserirDadosPessoa() em nossa classe exemplo Pessoa (figura 80).

O método showInputDialog() lê a informação e retorna esse dado para a linha na qual foi executado. No exemplo da figura 80, esse valor é passado por parâmetro para o método set, que o recebe e armazena nos atributos.

Para que o último exemplo fique mais claro, poderíamos reescrevê-lo da seguinte forma:

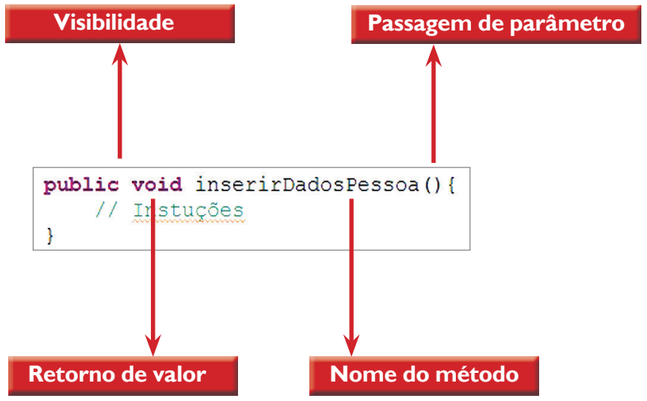
```
String vNome = JOptionPane.showInputDialog("Digite seu nome: ");
this.setNome( vNome );
```

Poderíamos armazenar o dado lido pelo showInputDialog() em uma variável do tipo String e, depois, passar essa variável de parâmetro para o método set, que, por sua vez, pegará esse dado e armazenará em seu respectivo atributo.

```
46    public void inserirDadosPessoa(){
47        this.setNome( JOptionPane.showInputDialog("Digite seu nome: ") );
48        this.setEndereco( JOptionPane.showInputDialog("Digite seu endereço: ") );
49        this.setTelefone( JOptionPane.showInputDialog("Digite seu telefone: ") );
50    }
```

Figura 80
Acréscimo do método inserirDadosPessoa().

Figura 81
Informações do cabeçalho de um método.



4.7. Assinatura de métodos

O cabeçalho de um método é composto por quatro informações (figura 81). As duas últimas (nome e passagem de parâmetros) definem sua assinatura. Quer dizer que não é possível ter dois métodos na mesma classe com exatamente o mesmo nome e a mesma configuração de passagem de parâmetros (quantidade e tipo de dado na mesma sequência). O Java e a orientação a objetos possibilitam o uso dessas características na construção dos métodos de maneira mais ampla, fazendo uso dos conceitos de sobrecarga, sobrescrita e herança.

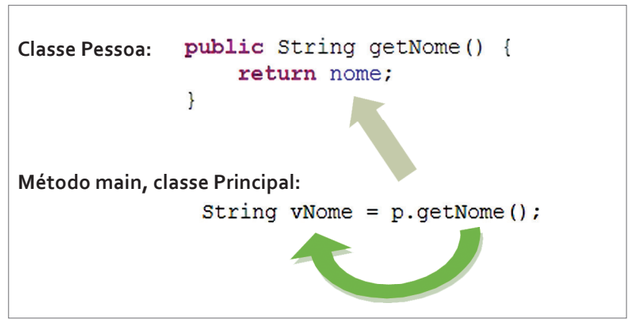
Agora, é importantíssimo entender a estrutura de um método. Toda a regra codificada (programada) em Java se localiza em métodos. Saber utilizar as possibilidades de obtenção (passagem de parâmetros) e de disponibilização (retorno) de dados nos métodos é vital não só para escrever novos códigos como também para compreender os já desenvolvidos.

No método inserirDadosPessoa(), exemplificado anteriormente, ele não retorna valor e não recebe parâmetros. Constatamos isso pelo uso da palavra "void" (vazio) e pela ausência de definições entre os parênteses "()". Concluímos, então, que se houver a necessidade de ler ou de apresentar informações, essas ações serão realizadas pelo próprio método. Em outras palavras, esse método não espera e não envia nenhuma informação para fora de seu escopo (definido pelas chaves "{ }"). Utilizaremos os métodos de acesso getNome() e setNome() como exemplos no detalhamento da entrada e saída de dados nos métodos.

4.7.1. Retorno de valor

O método getNome() lê o conteúdo do atributo nome e retorna essa informação para o local onde foi solicitada. O comando return envia o valor definido à sua frente (no caso, o conteúdo do atributo nome) para fora do método pela "porta de saída" de informações, definida com o tipo String.

Figura 82
Retornar um valor do tipo String.

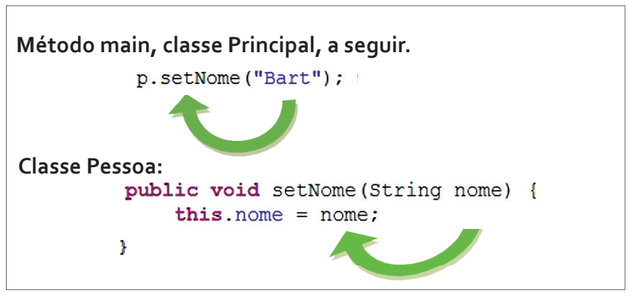


Portanto, o método está preparado para retornar um valor do tipo String e faz isso quando executa o comando return (figura 82).

4.7.2. Passagem de parâmetro

O método setNome foi escrito para receber um valor do tipo String e armazená-lo no atributo nome. A "porta de entrada" de valores para dentro do método são os parênteses, e essas informações (parâmetros) devem ser nomeadas e seus tipos definidos (figura 83).

Figura 83
Passagem de parâmetro.



4.7.3. Retorno de valor e passagem de parâmetro

Se for necessário, os métodos podem ser definidos para receber e retornar valor. Para exemplificar, modificaremos o método inserirDadosPessoa.

As três primeiras linhas armazenam os valores lidos em variáveis. Na quarta linha, usa-se o método inserirDadosPessoa do objeto p e as variáveis são passadas por parâmetro. Ainda nessa mesma linha, após a execução, o método inserirDadosPessoa retorna uma mensagem de confirmação armazenada na variável mensagem. E na quinta linha é apresentada a mensagem pelo método showMessageDialog(), como ilustra a figura 84.

Figura 84
Método main da Classe Principal.

```
String vNome = JOptionPane.showInputDialog("Digite seu nome: ");
String vEndereco = JOptionPane.showInputDialog("Digite seu endereço: ");
String vTelefone = JOptionPane.showInputDialog("Digite seu telefone: ");

String vMensagem = p.inserirDadosPessoa(vNome, vEndereco, vTelefone);

JOptionPane.showMessageDialog(null, vMensagem);
```

Figura 85
Classe Pessoa.

```
public String inserirDadosPessoa(String pNome, String pEndereco, String pTelefone){

    this.setNome( pNome );
    this.setEndereco( pEndereco );
    this.setTelefone( pTelefone );

    return "Inclusao de dados realizado com sucesso!";
}
```

O método recebe os três parâmetros e os passa (também por parâmetro) para os respectivos métodos sets, os quais armazenarão os dados nos atributos. Depois, retorna uma frase (String) de confirmação, como mostra a figura 85.

4.8. Estruturas e recursos da linguagem Java

4.8.1. Palavras reservadas do Java

As palavras reservadas do Java são aquelas que têm um sentido predeterminado nessa linguagem. Não podem ser usadas para outros fins, como por exemplo nomes de variáveis ou métodos. Confira quais são elas no quadro *Palavras Reservadas*.

PALAVRAS RESERVADAS

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

Segundo a Java Language Specification, as palavras null, true e false são chamadas de valores literais, e não de palavras reservadas, porém, assim como as palavras reservadas, não é possível utilizá-las para criar um identificador (variável, método etc.).

4.8.2. Tipos de dados

As informações manipuladas nos programas são chamadas de dados, os quais são organizados em tipos. Os tipos de dados básicos, também conhecidos como tipos primitivos, têm uma pequena variação entre as diferentes linguagens de programação (confira o quadro *Tipos primitivos de dados*).

TIPOS PRIMITIVOS DE DADOS

TIPO	ABRANGÊNCIA
boolean	Pode assumir o valor true (verdadeiro) ou o valor false (falso).
char	Caractere em notação Unicode de 16 bits. Serve para a armazenagem de dados alfanuméricos. Possui um valor mínimo de '\u0000' e um valor máximo de '\uffff'. Também pode ser usado como um dado inteiro com valores na faixa entre 0 e 65535.
byte	É um inteiro complemento de 2 em 8 bits com sinal. Ele tem um valor mínimo de -128 e um valor máximo de 127 (inclusive).
short	É um inteiro complemento de 2 em 16 bits com sinal. Possui um valor mínimo de -32.768 e máximo de 32.767 (inclusive).
int	É um inteiro complemento de 2 em 32 bits com sinal. Tem valor mínimo de -2.147.483,648 e máximo de 2.147.483,647 (inclusive).
long	É um inteiro complemento de 2 em 64 bits com sinal. O seu valor mínimo é -9.223.372.036.854.775,808 e o máximo, 9.223.372.036.854.775,807 (inclusive).
float	Representa números em notação de ponto flutuante normalizada em precisão simples de 32 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representado por esse tipo é 1.40239846e-46 e o maior, 3.40282347e+38
double	Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representado é 4.94065645841246544e-324 e o maior, 1.7976931348623157e+308

4.8.3. Operadores

ARITMÉTICOS	
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

RELACIONAIS	
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
==	Igual
!=	Diferente
instanceof	Comparação de tipo

LÓGICOS	
&&	E (and)
||	Ou (or)
!	Não (not)

BIT A BIT	
&	E (bitwise and)
|	Ou (bitwise or)
^	Ou exclusivo (bitwise xor)
<<	Deslocamento de bit à esquerda
>>	Deslocamento de bit à direita com extensão de sinal
>>>	Deslocamento de bit à direita sem extensão de sinal
~	Negação (inversão bit a bit)

ATRIBUIÇÕES	
=	Atribuição
+=	Atribuição com soma
-=	Atribuição com subtração
*=	Atribuição com multiplicação
/=	Atribuição com divisão

ATRIBUIÇÕES BIT A BIT	
&=	Atribuição de E bit a bit
|=	Atribuição de Ou bit a bit
^=	Atribuição de Ou exclusivo bit a bit
<<=	Atribuição de deslocamento de bit à esquerda
>>=	Atribuição de Deslocamento de bit à direita com extensão de sinal
>>>=	Atribuição Deslocamento de bit à direita sem extensão de sinal

TERNÁRIO	
?:	Condicional (if-then-else compacto)

INCREMENTO	
++	pré-incremento ou pós-incremento

DECREMENTO	
--	pré-decremento ou pós-decremento

4.8.4. Variáveis e constantes

Em Java, a declaração de variáveis é feita informando o tipo de dado seguido do nome da variável. Apesar de não ser obrigatório, em algumas situações é interessante inicializar os valores das variáveis no momento da declaração (figura 86).

Figura 86
Declaração de variáveis.

```
float a = 20f, b = 30f;
double x = 10d, y = 5.0;
int op = 2;
boolean status = true;
String nome = "";
```

Perceba que nos tipos float e double deve-se explicitar o formato. Caso isso não seja feito, ocorrerá uma promoção de tipo (visto a seguir). Constantes em Java são declaradas pelo comando final, ilustrado na figura 87.

Figura 87
Declaração de constantes.

```
final double pi = 3.1415;
```

4.8.5. Conversões de tipos de dados

4.8.5.1. Conversão

A transformação de um dado de um tipo em outro é realizada por métodos específicos, conforme o quadro *Conversão*.

Exemplo: uma variável do tipo String (figura 88).

Figura 88

```
String valorString = "100";
```

Convertida para dados numéricos (figura 89).

Figura 89

```
int valorInt = Integer.parseInt(valorString);
float valorFloat = Float.parseFloat(valorString);
double valorDouble = Double.parseDouble(valorString);
```

E variáveis do tipo numérico convertidas para String (figura 90).

Figura 90

```
valorString = String.valueOf(valorInt);
valorString = Float.toString(valorFloat);
valorString = Double.toString(valorDouble);
```


CONVERSÃO

De	Para	Métodos
String	int	Integer.parseInt()
String	float	Float.parseFloat()
String	double	Double.parseDouble()
int	String	Integer.toString() ou String.valueOf()
float	String	Float.toString() ou String.valueOf()
double	String	Double.toString() ou String.valueOf

4.8.5.2. Cast (matriz)

São conversões explícitas entre tipos primitivos, sempre respeitando a faixa de abrangência de cada tipo. Basicamente, o cast é feito do tipo "maior" para o "menor" sob risco de truncamento do valor se essa ordem não for respeitada (observe o quadro *Cast*).

CAST

Tipos	Cast válidos
byte	nenhuma
short	byte
char	byte e short
int	byte, short e char
long	byte, short, char e int
float	byte, short, char, int e long
double	byte, short, char, int, long e float

Exemplos (figura 91).

Figura 91

```
byte varByte = 10;
short varShort;
long varLong;

varShort = (short) varByte;
varLong = (long) varShort;
```

4.8.5.3. Promoção

A promoção pode ser entendida como um tipo de cast automático (implícito), conforme mostra o quadro *Promoção*.

PROMOÇÃO

Tipos	Promoções válidas
double	nenhuma
float	double
long	float ou double
int	long, float ou double
char	int, long, float ou double
short	int, long, float ou double (char não)
byte	short, int, long, float ou double (char não)
boolean	nenhuma

Veja alguns exemplos na figura 92.

Figura 92

```
int varInt = 20;
float varFloat;
double varDouble;

varFloat = varInt;
varDouble = varFloat;
```


4.8.6. Desvios condicionais

4.8.6.1. If-else

A estrutura de if é extremamente adaptável, podendo assumir qualquer uma das formas ilustradas nas figuras 93, 94 e 95.

Figura 93

```
if(x > y){
    JOptionPane.showMessageDialog(null, "X é maior que Y");
}
if(x >= y){
    JOptionPane.showMessageDialog(null, "X é maior ou igual a Y");
}else{
    JOptionPane.showMessageDialog(null, "Y é maior que X");
}
```

Figura 94

```
if(x > y){
    JOptionPane.showMessageDialog(null, "X é maior que Y");
}else{
    if(y > x){
        JOptionPane.showMessageDialog(null, "Y é maior que X");
    }else{
        JOptionPane.showMessageDialog(null, "X e Y são iguais");
    }
}
```

Figura 95

```
if(x > y){
    JOptionPane.showMessageDialog(null, "X é maior que Y");
}else if(y > x){
    JOptionPane.showMessageDialog(null, "Y é maior que X");
}else if(x == y){
    JOptionPane.showMessageDialog(null, "X e Y são iguais");
}
```

O if considera o valor lógico (true ou false, literalmente verdadeiro ou falso) resultante da condição. Se o tipo testado for um boolean, podemos aproveitar seu próprio formato como condição (figura 96).

Figura 96

```
if(status){
    JOptionPane.showMessageDialog(null, "Status verdadeiro");
}else{
    JOptionPane.showMessageDialog(null, "Status falso");
}
```

Quando as classes assumem o papel de tipos, como é o caso do String, esses objetos possuem métodos para manipulação de seus conteúdos. Por exemplo, a comparação de Strings (textos) é feita pelo método equals() que retorna true, se o valor passado por parâmetro for igual ao conteúdo armazenado, e false, se for diferente (figura 97).

Figura 97

```
if(nome.equals("Aluno")){
    JOptionPane.showMessageDialog(null, "Bom dia " + nome);
}else{
    JOptionPane.showMessageDialog(null, "Nome desconhecido");
}
```

Outras classes são utilizadas como tipos (por exemplo, Date) e, visualmente, são de fácil identificação pela primeira letra maiúscula (tipos primitivos sempre começam com letra minúscula).

4.8.6.2. Switch-case

Em Java, o switch-case só aceita a condição nos tipos int, byte e char (figura 98).

Figura 98
Switch case.

```
switch (op) {
case 1:
    JOptionPane.showMessageDialog(null, "Opção 1 escolhida!");
    break;

case 2:
    JOptionPane.showMessageDialog(null, "Opção 2 escolhida!");
    break;

case 3:
    JOptionPane.showMessageDialog(null, "Opção 3 escolhida!");
    break;

default:
    JOptionPane.showMessageDialog(null, "Nenhuma das opções foram escolhidas!");
    break;
}
```

4.8.7. Laços de repetição

4.8.7.1. While

O teste condicional está no começo do looping, ou seja, se a condição não for verdadeira, o bloco de código contido no while não será executado. Se for, esse bloco será repetido enquanto a condição permanecer verdadeira (figura 99).

Figura 99

```
op = 1;
while(op != 0){
    op = Integer.parseInt(JOptionPane.showInputDialog("Digite uma opção"));
    JOptionPane.showMessageDialog(null, "Opção digitada: " + op);
}
```

4.8.7.2. Do-while

O teste condicional está no final do looping, isto é, o bloco de código contido no do-while será executado a primeira vez e, enquanto a condição for verdadeira, será repetido (figura 100).

Figura 100

```
do{
    op = Integer.parseInt(JOptionPane.showInputDialog("Digite uma opção"));
    JOptionPane.showMessageDialog(null, "Opção digitada: " + op);
}while(op != 0);
```

4.8.7.3. For

O bloco de código contido no “for” será repetido quantas vezes a condição na abertura do looping determinar. No exemplo em seguida, a variável irá de 0 a 9 (figura 101).

Figura 101

```
for(int i=0 ; i < 10 ; i++){
    JOptionPane.showMessageDialog(null, "Contador = " + i + "\n");
}
```

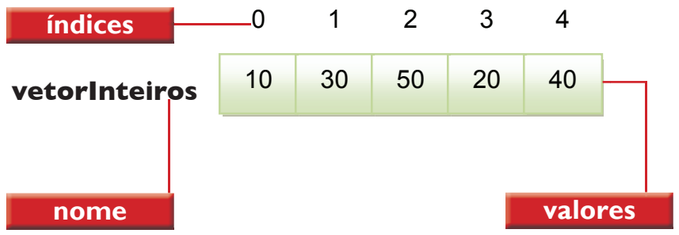
4.8.8. Array (Vetor)

Um array, também chamado de vetor, é uma estrutura parecida com as variáveis, porém, que permite armazenar vários valores, desde que sejam do mesmo tipo. Por isso, pode ser definido como uma estrutura homogênea de dados. Ele pode ser de tipos unidimensional e bidimensional. A quantidade de valores que um array consegue armazenar é definida na sua declaração.

4.8.8.1. Unidimensional

Um array pode ser composto por uma única dimensão (única linha), conforme mostra a figura 102.

Figura 102
Vetor unidimensional.



Exemplos:
A declaração pode ser feita de três maneiras:
Em duas etapas (primeiro, definindo-se o tipo e, depois a dimensão, conforme a figura 103).

Figura 103

```
double[] vetorReais;
vetorReais = new double[10];
```

Em uma única linha (figura 104).

Figura 104

```
int[] vetorInteiros = new int[5];
```

Atribuindo valores às posições (figura 105).

Figura 105

```
String[] paises = {"Brasil", "Inglaterra", "Espanha"};
```

Essa atribuição pode ser feita de duas maneiras:
Indicando explicitamente o índice (posição) desejado (figura 106).

Figura 106

```
vetorInteiros[0] = 10;
vetorInteiros[1] = 30;
vetorInteiros[2] = 50;
vetorInteiros[3] = 20;
vetorInteiros[4] = 40;
```


Por meio de loopings, utilizando um contador (figura 107).

Figura 107

```
for(int i=0; i<5; i++){
    vetorInteiros[i] = Integer.parseInt(
                       JOptionPane.showInputDialog("Digite um valor:")
                       );
}
```

A leitura (apresentação) pode ser feita de três maneiras:
Acessando explicitamente o índice (posição) desejado (figura 108).

Figura 108

```
JOptionPane.showMessageDialog(null, "O 1° valor é: " + vetorInteiros[0]);
JOptionPane.showMessageDialog(null, "O 2° valor é: " + vetorInteiros[1]);
JOptionPane.showMessageDialog(null, "O 3° valor é: " + vetorInteiros[2]);
JOptionPane.showMessageDialog(null, "O 4° valor é: " + vetorInteiros[3]);
JOptionPane.showMessageDialog(null, "O 5° valor é: " + vetorInteiros[4]);
```

Por loopings, utilizando um contador (figura 109).

Figura 109

```
for(int i=0; i<5; i++){
    JOptionPane.showMessageDialog(
        null, "O " + (i+1) + "° valor é: " + vetorInteiros[i]
    );
}
```

Usando um iterador (detalhado adiante), como na figura 110.

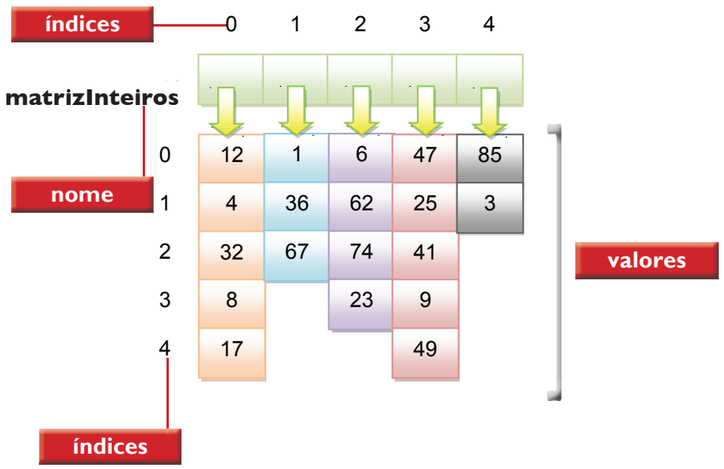
Figura 110

```
for(String p : paises){
    JOptionPane.showMessageDialog(null, "Paises: " + p);
}
```

4.8.8.2. Bidimensional (Matrizes)

Um array bidimensional (ou multidimensional) em Java pode ser entendido como um array unidimensional cujas posições contêm outros arrays unidimensionais e podem ser regulares (mesma quantidade de linhas para todas as colunas) ou irregulares (quantidade de linhas diferente para cada coluna), conforme figura 111.

Figura 111
Vetor bidimensional irregular.



Exemplos:
A declaração pode ser feita de três maneiras:
Em duas etapas (primeiro, definindo-se o tipo e, depois, a dimensão), como na figura 112.

Figura 112

```
double[][] matrizDouble;
matrizDouble = new double[3][3];
```

Em uma única linha, como na figura 113.

Figura 113

```
float[][] matrizFloat = new float[6][4];
```

Atribuindo valores às posições (figura 114).

Figura 114

```
String[][] paisesCapitais = { {"Brasil", "Inglaterra", "Espanha"},
                              {"Brasilia", "Londres", "Madri"   } };
```


Já a atribuição pode ser realizada de duas formas:

Indicando explicitamente o índice (posição) desejado (figura 115).

Figura 115

```
matrizDouble[0][0] = 10;
matrizDouble[0][1] = 20;
matrizDouble[0][2] = 30;

matrizDouble[1][0] = 40;
matrizDouble[1][1] = 50;
matrizDouble[1][2] = 60;

matrizDouble[2][0] = 70;
matrizDouble[2][1] = 80;
matrizDouble[2][2] = 90;
```

Por meio de loopings, utilizando um contador (figura 116).

Figura 116

```
for(int linha=0; linha < 3; linha++){

    for(int coluna=0; coluna < 3; coluna++){

        matrizDouble[linha][coluna] = Double.parseDouble(
                                        JOptionPane.showInputDialog(
                                            "Digite uma valor para a posição " +
                                            "[" + linha + "][" + coluna + "]"
                                        )
                                      );
    }
}
```

Quanto à leitura (apresentação), há duas opções para fazê-la:

Acessando explicitamente o índice (posição) desejado (figura 117).

Figura 117

```
JOptionPane.showMessageDialog(null, "linha 0 coluna 0: " + matrizDouble[0][0]);
JOptionPane.showMessageDialog(null, "linha 0 coluna 1: " + matrizDouble[0][1]);
JOptionPane.showMessageDialog(null, "linha 0 coluna 2: " + matrizDouble[0][2]);

JOptionPane.showMessageDialog(null, "linha 1 coluna 0: " + matrizDouble[1][0]);
JOptionPane.showMessageDialog(null, "linha 1 coluna 1: " + matrizDouble[1][1]);
JOptionPane.showMessageDialog(null, "linha 1 coluna 2: " + matrizDouble[1][2]);

JOptionPane.showMessageDialog(null, "linha 2 coluna 0: " + matrizDouble[2][0]);
JOptionPane.showMessageDialog(null, "linha 2 coluna 1: " + matrizDouble[2][1]);
JOptionPane.showMessageDialog(null, "linha 2 coluna 2: " + matrizDouble[2][2]);
```

Por meio de loopings, utilizando um contador (figura 118).

Em um dos exemplos anteriores, foi manipulada uma matriz de 3 x 3, do tipo double, que pode ser representada pela imagem da figura 119.

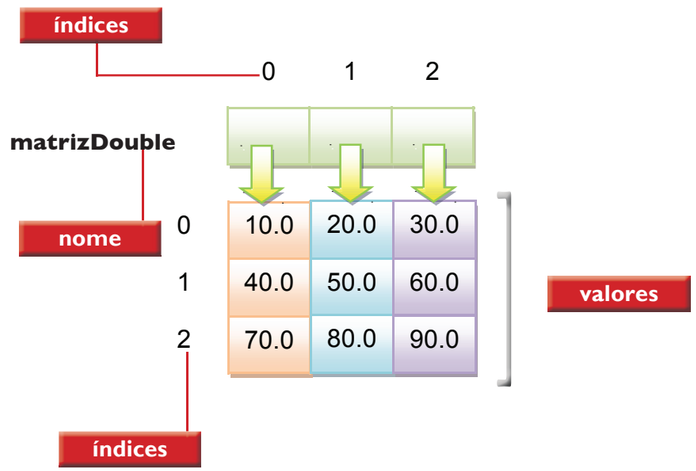
Figura 118

```
for(int linha=0; linha < 3; linha++){

    for(int coluna=0; coluna < 3; coluna++){

        JOptionPane.showMessageDialog(null, "linha " + linha + " " +
                                            "coluna " + coluna + ": " +
                                            matrizDouble[linha][coluna]
        );
    }
}
```

Figura 119
Vetor bidimensional quadrado.



4.8.9. Collections

É um conjunto de classes e interfaces disponíveis no pacote java.util, que faz o uso do import necessário. Essas classes fornecem recursos para trabalhar com conjuntos de elementos (coleções). Na verdade, todas as funcionalidades encontradas nas collections poderiam ser feitas manualmente, com a utilização de recursos normais da linguagem. A vantagem das collections é que elas já trazem essas funcionalidades prontas.

O estudo das collections é relativamente extenso, pois existem muitos recursos disponíveis. Por enquanto, vamos conhecer uma dessas coleções, a ArrayList, responsável pelo manuseio de arrays. Sua grande vantagem é contar com métodos prontos.

4.8.9.1. ArrayList

Antes de nos aprofundarmos nessa coleção, vamos primeiro conhecer melhor a interface List. Como o próprio nome indica, ela cria e gerencia uma lista. Isso significa que a ordem de inserção será mantida. A interface List mantém seus elementos indexados. Permite, portanto, que exista uma preocupação com o posicionamento de cada elemento e que tal posição seja determinada pelo índice. Devemos utilizar uma List quando a ordem de inserção ou a posição na coleção nos interessa. Há alguns métodos importantes, quando manipulamos Lists (confira o quadro *Métodos da Interface List*).

MÉTODOS DA INTERFACE LIST

Método	Descrição
add (objeto)	Adiciona um elemento à List na última posição
get (índice)	Retorna o elemento da posição do índice
iterator ()	Retorna um objeto do tipo Iterator
size ()	Retorna um int (inteiro) com a quantidade de elementos da coleção
contains (objeto)	Retorna true se o elemento já existe dentro do List
clear ()	Elimina todos os elementos do List

Um ArrayList pode ser entendido como um array dinâmico: ele aumenta ou diminui sua dimensão à medida em que é utilizado. Além disso, é possível armazenar dados (e objetos) de diferentes tipos no mesmo ArrayList. A classe ArrayList é filha (subclasse) de List e, portanto, tem todos os métodos pertencentes à List. Veremos mais sobre subclasses adiante, quando estudarmos o conceito de herança.

Para fechar essa introdução básica sobre Collections, vamos ver, em seguida, alguns métodos muito úteis na manipulação de ArrayList (observe o quadro *Métodos da Collection ArrayList)*.

MÉTODOS DA COLLECTION ARRAYLIST

Método	Descrição
sort (ArrayList)	Ordena os elementos em ordem crescente
shuffle (ArrayList)	É o oposto do sort. Ao invés de ordenar, ele desordena (mistura) os elementos do ArrayList
binarySearch (ArrayList, "valor")	Pesquisa um valor no ArrayList e retorna sua posição (índice). Se não for encontrado, retorna um número inteiro negativo
reverse (ArrayList)	Inverte a ordem dos elementos
frequency (ArrayList, "valor")	Conta a quantidade de ocorrências do elemento especificado

Exemplos:
A declaração pode ser feita sem a definição de tipo de dado a ser armazenado e, com isso, qualquer tipo é aceito (figura 120).

Figura 120

```
String descricao = "Produto";
double preco = 100d;
int qtdeEstoque = 30;

List arrayListExemplo = new ArrayList();

arrayListExemplo.add(descricao);
arrayListExemplo.add(preco);
arrayListExemplo.add(qtdeEstoque);


for(Object dado : arrayListExemplo){
    JOptionPane.showMessageDialog(null, "Conteúdo: " + dado);
}
```

Saída obtida (figura 121).

Figura 121

A definição do tipo de dado é feita pelos delimitadores <>, e as chamadas aos métodos podem ser feitas como nos exemplos adiante.

Declaração de um ArrayList do tipo int (figura 122).

Figura 122

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


Atribuição de valores (figura 123).

Figura 123

```
arrayInteiros.add(30);
arrayInteiros.add(50);
arrayInteiros.add(10);
```

Ordenação crescente (figura 124).

Figura 124

```
Collections.sort(arrayInteiros);
```

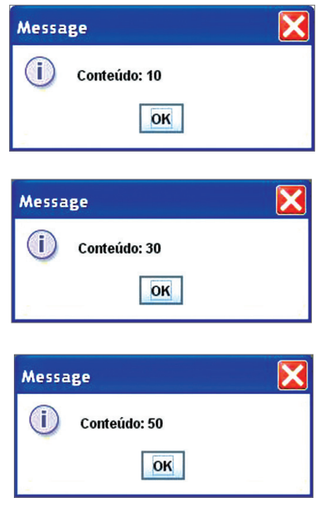
Listagem do arrayInteiros já ordenado por um iterator (dado), como na figura 125.

Figura 125

```
for(int dado : arrayInteiros){
    JOptionPane.showMessageDialog(null, "Conteúdo: " + dado);
}
```

Saída obtida (figura 126).

Figura 126



Pesquisa se o número 30 consta no arrayInteiros (figura 127).

Figura 127

```
int resultado = Collections.binarySearch(arrayInteiros, 30);
```

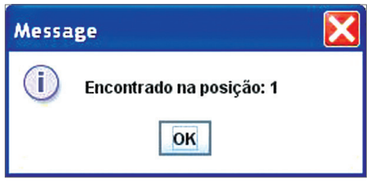
Verificando o resultado da pesquisa (figura 128).

Figura 128

```
if(resultado > 0){
    JOptionPane.showMessageDialog(null, "Encontrado na posição: " + resultado);
}else{
    JOptionPane.showMessageDialog(null, "Valor não encontrado!");
}
```

Resultado obtido (figura 129).

Figura 129



4.9. Orientação a objetos (2)

Uma vez estudados os fundamentos e sintaxes básicas para a construção de objetos, e que devem ser muito bem assimilados, o objeto de estudo, em seguida, são os principais conceitos de orientação a objetos, detalhando suas características e aplicabilidades.

4.9.1. 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 (solicitações que o sistema deverá atender), podemos destacar os atributos ou os métodos comuns a um grupo de classes e concentrá-los em uma única classe (processo conhecido como generalização). Da mesma forma, é possível identificar o que é pertinente somente a determinada classe (conhecido como especificação). A primeira vantagem dessa organização é evitar a duplicidade de código (ter o mesmo trecho de código em lugares diferentes do sistema), o que traz maior agilidade e confiabilidade na manutenção e expansão do sistema.

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. Observe as definições de clientes, fornecedores e funcionários utilizados na livraria de acordo com o diagrama de classes (figura 130).

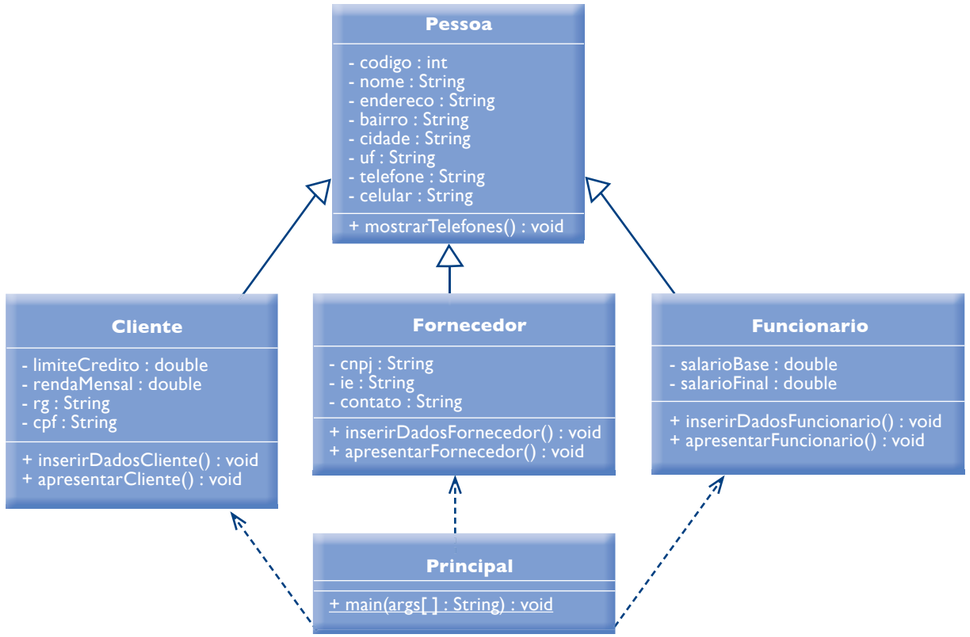


Figura 130
Herança.

A classe pessoa contém oito atributos e um método, os quais são comuns para clientes, fornecedores e funcionários e, portanto, deveriam constar nas classes Cliente, Fornecedor e Funcionario. Sem o recurso da herança, teríamos que replicar esses atributos e métodos nas três classes, procedimento desaconselhável em qualquer linguagem de programação, por trazer complexidades extras na manutenção e expansão dos sistemas. Por exemplo, vamos considerar um método para emissão de correspondência que foi atualizado para começar a gerar um histórico de remessas. Tal atualização deveria ser feita nas três classes envolvidas e, caso uma delas não fosse realizada, a atualização do controle de remessas (geração de histórico) ficaria inconsistente.

No modelo acima, a classe Pessoa foi definida como uma superclasse e as classes Cliente, Fornecedor e Funcionario, como suas subclasses. Do ponto de vista da funcionalidade, tudo o que foi definido na superclasse (atributos e métodos) será herdado pelas suas subclasses. Ou seja, um objeto instanciado a partir da classe Cliente possui 12 atributos. São eles: nome, endereço, bairro, cidade, UF, telefone e celular declarados na superclasse Pessoa, além de limiteCredito, rendaMensal, RG e CPF na subclasse Cliente. Há ainda três métodos, nos quais mostrar telefones foi definido na classe Pessoa e inserir DadosCliente e apresentarCliente foram definidos na classe Cliente. Na utilização desses atributos e métodos para um objeto do tipo Cliente, fica transparente o local onde cada um foi declarado ou definido.

Para estabelecer a herança em relação à codificação, as superclasses continuam com a mesma estrutura de uma classe comum (como vimos em exemplos anteriores). Já as subclasses recebem as seguintes definições:

Abertura da classe (figura 131).

Figura 131

```
public class Cliente extends Pessoa {
```

O comando extends é o responsável por estabelecer a herança. É inserido na abertura da subclasse e indica o nome da superclasse, criando vínculo entre elas.

Construtores (figura 132).

Figura 132

```
public Cliente() {
    this(0,"","","","","","","",0,0,"","");
}

public Cliente(int codigo, String nome, String endereco, String bairro,
               String cidade, String uf, String telefone, String celular,
               double limiteCredito, double rendaMensal, String rg, String cpf) {

    super(codigo, nome, endereco, bairro, cidade, uf, telefone, celular);

    this.limiteCredito = limiteCredito;
    this.rendaMensal = rendaMensal;
    this.rg = rg;
    this.cpf = cpf;
}
```

Os construtores estão diretamente relacionados à inicialização dos atributos de uma classe. Partindo desse princípio e considerando o nosso exemplo, um objeto do tipo cliente possui todos os atributos declarados na sua superclasse (Pessoa) mais os declarados na classe Cliente. Portanto, o construtor de uma subclasse deve estar preparado para inicializar os atributos herdados e os declarados na própria classe.

No construtor que recebe parâmetros (aquele que inicializa os atributos com algum valor), utilizamos o método super() para invocar o construtor da superclasse. Isso porque já foi definida nele a forma como esses atributos serão inicializados (reutilizando o construtor já existente na superclasse), restando apenas inicializar os atributos na subclasse.

Para acessar os atributos da superclasse, obrigatoriamente, devemos utilizar seus métodos de acesso (getters e setters), ao contrário dos atributos instanciados na própria classe, que podem ser acessados diretamente. Porém, como já comentamos antes, para garantir o conceito de encapsulamento e usufruir de seus benefícios (segurança, manutenabilidade etc.), sempre devemos criar e utilizar os métodos getters e setters para todos os atributos. Em relação ao nosso exemplo, o mesmo se aplica às classes Fornecedor e Funcionario.

4.9.2. 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

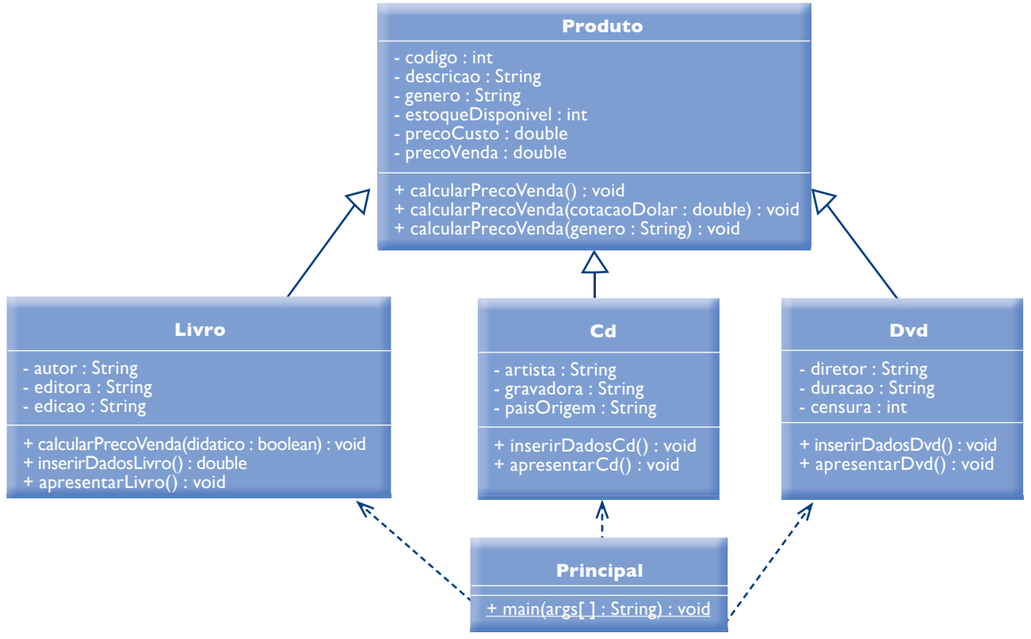


Figura 133
Sobrecarga do método calcularPrecoVenda.

livraria são livros, CDs e DVDs, como representado no diagrama de classe na figura 133.

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. Observe, na figura 134, como fica a codificação dos métodos na classe Produto.

Figura 134
Codificação dos métodos na classe Produto.

```
public void calcularPrecoVenda(){
    this.setPrecoVenda(this.getPrecoCusto() + (this.getPrecoCusto() * 0.2));
}


public void calcularPrecoVenda(double cotacaoDolar){
    this.setPrecoVenda(this.getPrecoCusto() * cotacaoDolar);
}


public void calcularPrecoVenda(String genero){

    if(this.getGenero().equals(genero)){
        this.setPrecoVenda(this.getPrecoCusto() + (this.getPrecoCusto() * 0.2));
    }
}
```

E na classe Livro (figura 135):

Figura 135
Codificação do métodos na classe Livro.

```
public void calcularPrecoVenda(boolean didatico){

    if(didatico){
        this.setPrecoVenda(this.getPrecoCusto() + (this.getPrecoCusto() * 0.1));
    }
}
```

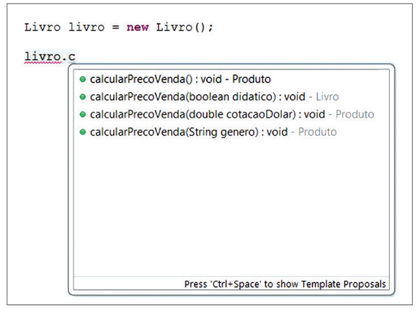
Os pontos importantes na codificação anterior são:

- A diferenciação das assinaturas dos métodos se baseia na quantidade e no tipo de dado dos parâmetros.

- A sobrecarga pode ser realizada na mesma classe ou em subclasses, e os conceitos de herança são aplicados na utilização dos objetos. Em nosso exemplo, um objeto do tipo Livro tem quatro métodos calcularPrecoVenda(), já CD e DVD têm três.

No método main, teremos o que ilustra a figura 136.

Figura 136



O Java identificará o método que deve ser executado de acordo com a chamada realizada, como mostra o exemplo da figura 137.

Figura 137

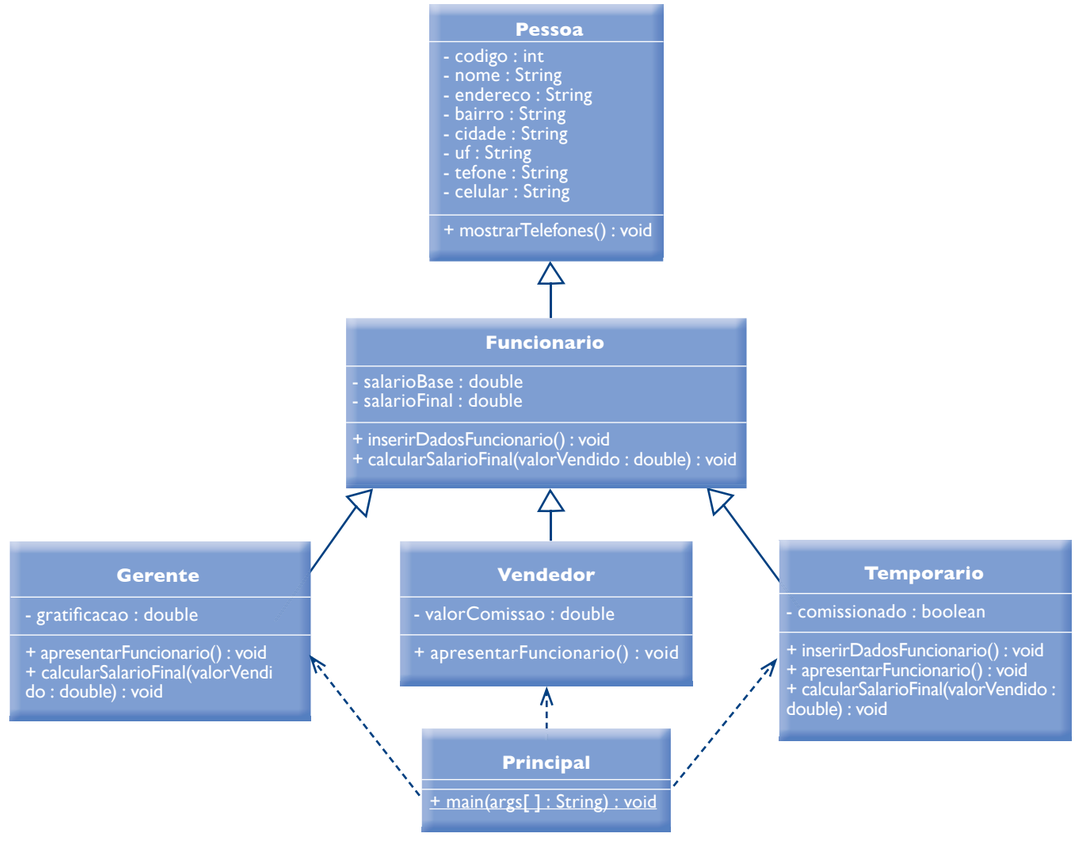
```
livro.calcularPrecoVenda(1.9);
```

Como está sendo passado um valor do tipo double, será executado o método calcularPrecoVenda que considera a cotação do dólar.

4.9.3. 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 da livraria (figura 138).

Figura 138
Sobrescrita do método calcularSalarioFinal.



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. Na classe Funcionario, o método foi codificado da forma como ilustra a figura 139.

Figura 139

```
public void calcularSalarioFinal(double valorVendido){
    this.setSalarioFinal( this.getSalarioBase() + (valorVendido * 0.1) );
}
```

Já na subclasse Gerente fica da seguinte maneira (figura 140).

Figura 140

```
public void calcularSalarioFinal(double valorVendido){

    double meta = Double.parseDouble( JOptionPane.showInputDialog(
                                        "Digite a meta de vendas do mês: "
                                      )
                );

    if(valorVendido > meta){
        this.setGratificacao(valorVendido * 0.15);
    }else{
        this.setGratificacao(0);
    }

    this.setSalarioFinal( this.getSalarioBase() + this.getGratificacao() );
}
```

E na subclasse Temporario, veja a figura 141.

Figura 141

```
public void calcularSalarioFinal(double valorVendido){

    if(this.comissionado){
        this.setSalarioFinal( this.getSalarioBase() + valorVendido * 0.1 );
    }else{
        this.setSalarioFinal( this.getSalarioBase() );
    }
}
```

Os objetos que representarão um gerente, um vendedor e um temporário serão instanciados a partir das classes Gerente, Vendedor e Temporario. Nesse momento (da criação dos objetos), o Java considera a estrutura de cada subclasse, a qual dá origem ao objeto. Então, um objeto do tipo Gerente considera a codificação do método calcularSalarioFinal da classe Gerente; um do tipo Temporario leva em conta a codificação da classe Temporario, e um do tipo Vendedor focaliza o método herdado, mas todos são invocados exatamente da mesma forma (figura 142).

A sobrescrita de métodos também proporciona vantagens relacionadas ao gerenciamento polimórfico de objetos.


```
Gerente gerente = new Gerente();
Vendedor vendedor = new Vendedor();
Temporario temporario = new Temporario();

gerente.calcularSalarioFinal(5000.00);
vendedor.calcularSalarioFinal(5000.00);
temporario.calcularSalarioFinal(5000.00);
```

4.9.4. Classes e métodos abstratos

É preciso, a princípio, conhecer os recursos utilizados na modelagem de sistemas. Para entender melhor o assunto, devemos ampliar o horizonte e pensar na arquitetura e no desenho da aplicação como um todo, assim como na manutenção e na expansão, atividades que podem ser realizadas por programadores (ou equipes de programadores) distintas e em momentos diferentes, mas que sempre seguirão as definições descritas no projeto. A primeira abordagem de **classe abstrata** é que se trata de classes que não darão origem a objetos. Em outras palavras, um objeto não poderá ser instanciado a partir delas. A sua finalidade, então, é definir, por generalização (possibilitada pela herança), os recursos (atributos e métodos) comuns a um grupo de classes (subclasses). Analisemos a representação dos clientes existentes na livraria no diagrama da figura 143.

Figura 142

As classes e métodos que vimos anteriormente são chamados de concretos. Os que estamos estudando agora são as classes e métodos abstratos, que possuem algumas características específicas: 1. Em UML, classes e métodos abstratos são formatados em itálico. 2. Uma classe concreta só pode conter métodos concretos. 3. Uma abstrata pode conter métodos concretos e abstratos. 4. Uma subclasse que herda um método abstrato é obrigada a incluir a assinatura do método, contendo ou não implementação (programação). 5. Enquanto não for incluída a assinatura do método abstrato herdado, a subclasse apresentará um erro que impedirá sua compilação.

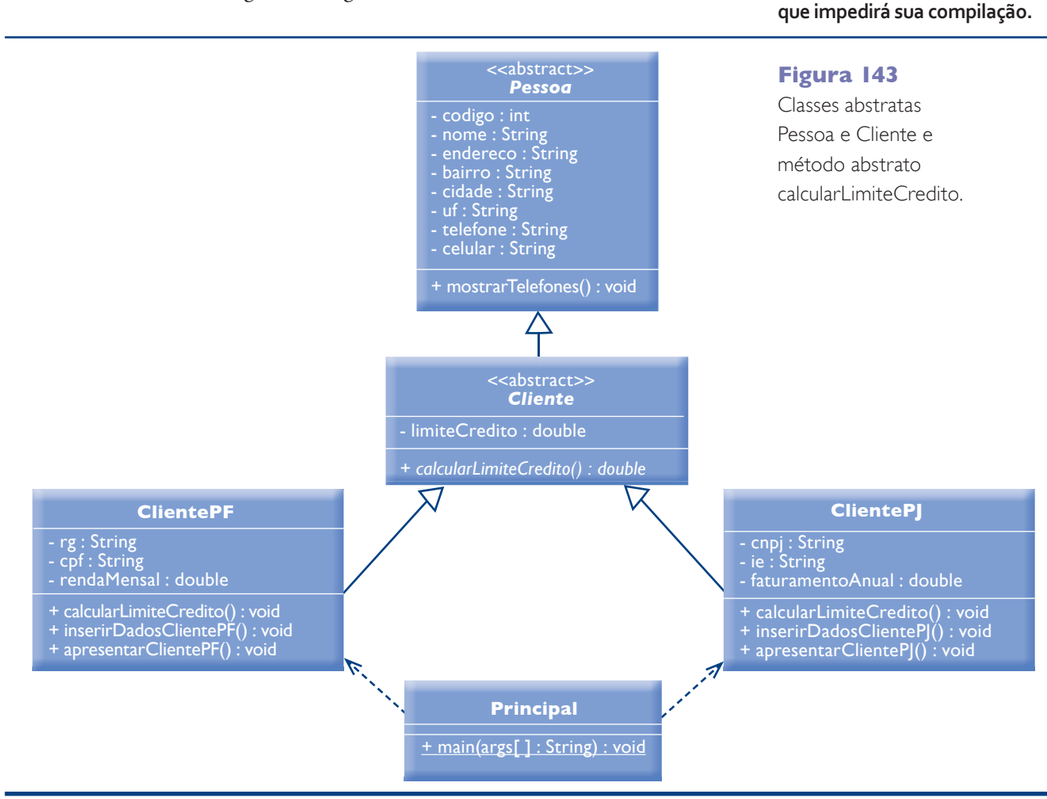


Figura 143
Classes abstratas Pessoa e Cliente e método abstrato calcularLimiteCredito.

As classes Cliente e Pessoa, nesse contexto, servem somente para definir quais e como serão os recursos (atributos e métodos) comuns a ClientePF e ClientePJ. Os objetos que representarão os clientes da livraria serão instanciados a partir das classes ClientePF e ClientePJ e nunca de Cliente e muito menos de Pessoa. Ao definir as classes Pessoa e Cliente como abstratas, evitamos que, por uma eventual falha de desenvolvimento, objetos sejam instanciados a partir delas, o que não faria sentido. Além de classes, podemos também definir métodos como abstratos, os quais servem para definir e forçar, uma padronização nas subclasses em relação à existência e à assinatura de métodos herdados.

Um método abstrato não possui implementação (codificação). É composto somente pela assinatura, na qual são definidos seu nome, passagem de parâmetros e retorno de valor. Na codificação de uma subclasse que estende de uma superclasse que, por sua vez, possui um método abstrato, o programador é obrigado a inserir, pelo menos, a assinatura desse método, podendo ou não implementá-lo (codificá-lo). Voltando ao exemplo dos clientes da livraria apresentado na figura 130, calcularLimiteCredito definido na classe Cliente é um método abstrato. Dessa forma, nas classes ClientePF e ClientePJ, somos obrigados a incluí-lo. A declaração abstract define uma classe abstrata, conforme se observa na figura 144.

Figura 144

```
abstract class Pessoa {
abstract class Cliente extends Pessoa {
```

A declaração abstract também é usada para métodos (figura 145).

Figura 145

```
public abstract void calcularLimiteCredito();
```

A utilização de um método abstrato garante que todas as subclasses de Cliente obrigatoriamente terão, pelo menos, a assinatura do método calcularLimiteCredito. Isso garante não só a padronização de modelagem (os programadores serão obrigados a seguir tais definições) como também a aplicação de conceitos polimórficos.

4.9.5. 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 as classes que as "assinam" (implementam) se comprometem a seguir, ou seja, devem seguir os métodos nela definidos. A figura 146 ilustra uma implementação de interface.

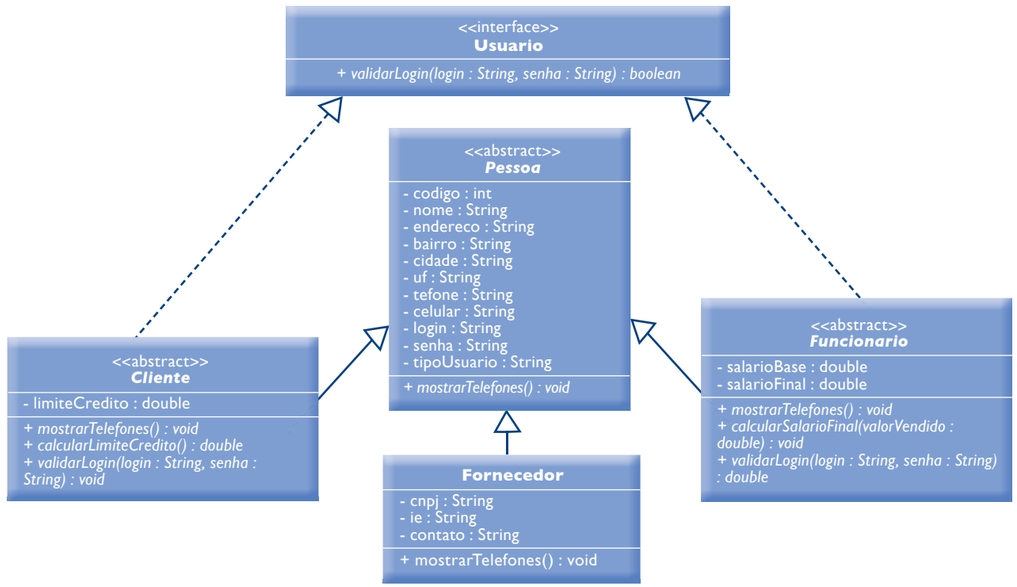


Figura 146
Implementação da inteface Usuario pelas classes Cliente e Funcionario.

Cliente e Funcionario

Para utilizar o sistema da livraria, o usuário deverá ser identificado por um login e uma senha. Os funcionários terão acesso ao sistema para utilizá-lo em suas rotinas de trabalho, como cadastros e atualizações de dados dos clientes, produtos, movimentação, entre outros recursos disponíveis no sistema, cujo acesso também será permitido aos clientes, para que possam visualizar promoções especiais e também entrar em contato com a livraria, pelo seu site na internet. Assim, tanto funcionários como clientes deverão ser devidamente identificados, para ter acesso ao sistema, algo que será realizado da mesma forma (recebendo o login e a senha e retornando verdadeiro ou falso). Então, pode-se concluir que o ideal seria criar um método de validação utilizado tanto por clientes como por funcionários. Mas para fazer isso nos deparamos com algumas dificuldades:

- Cliente e Funcionario são subclasses de Pessoa. Assim, se incluíssemos o método de validação de usuário na superclasse Pessoa, os dois herdariam esse método e o nosso problema estaria aparentemente resolvido. Porém, a classe Fornecedor também é subclasse de Pessoa e, por isso, também herdaria o método de validação. Só que os fornecedores não deverão acessar o sistema.

- Se o método de validação for incluído na superclasse Cliente, suas subclasses possuiriam o método de validação. Todavia, os funcionários pertencem a outro nível hierárquico de classe (a partir da subclasse Funcionario) e não teriam acesso ao método e validação.

- Da mesma forma, se forem incluídos na classe Funcionario, os clientes também não terão acesso.

O uso de uma interface resolve o problema já que, para implementá-la, as classes não precisam ser da mesma hierarquia. Lembre-se que uma interface só possui métodos abstratos. Assim, a classe Cliente e a classe Funcionario não receberão o método de validação já codificado. Entretanto, a partir dessa implementação, teremos a certeza de que existirá um método (com a mesma assinatura) nas classes (e subclasses) Cliente e Funcionario. Veja a codificação da interface Usuario na figura 147.

Figura 147

```
public interface Usuario {

    public void validarLogin(String login, String senha);

}
```

Para interfaces, não é obrigatório o uso do comando abstract, ficando implícito que todos os métodos inseridos em uma interface são abstratos. A "assinatura" de uma interface é realizada pelo comando implements da seguinte forma:

Classe Cliente (figura 148).

Figura 148

```
abstract class Cliente extends Pessoa implements Usuario {
```

Classe ClientePF (figura 149).

Figura 149

```
public void validarLogin(String login, String senha) {

    if(this.getLogin().equals(login) && this.getSenha().equals(senha)){
        JOptionPane.showMessageDialog(null,
                "Usuário cliente pessoa física autorizado!"
        );

    }else{
        JOptionPane.showMessageDialog(null,
                "Usuário cliente pessoa física não autorizado!"
```

Classe Funcionario (figura 150).

Figura 150

```
abstract class Funcionario extends Pessoa implements Usuario {
```

Classe Gerente (figura 151).

Figura 151

```
public void validarLogin(String login, String senha) {

    if( this.getLogin().equals(login) && this.getSenha().equals(senha) ){

        JOptionPane.showMessageDialog(null,
                "Usuário gerente autorizado!"
        );

    }else{
        JOptionPane.showMessageDialog(null,
                "Usuário gerente não autorizado!"
```

Outra importante característica das interfaces é que uma mesma classe pode implementar várias delas. Como não existe herança múltipla em Java (uma subclasse ter mais de uma superclasse), o uso de interface supre essa eventual necessidade. Por exemplo, para calcular os impostos a serem pagos pela livraria, devemos considerar que existem aqueles que incidem sobre os funcionários e os que incidem sobre os produtos comercializados. Nesse caso, podemos criar uma interface chamada Custos que defina um método calcularImpostos, o qual deverá ser implementado na classe Funcionario e na classe Produto. Veja o exemplo da classe Funcionario na figura 152.

Figura 152

```
abstract class Funcionario extends Pessoa implements Usuario, Custos{
```

4.9.6. Polimorfismo

O polimorfismo é a possibilidade de utilizar um objeto "como se fosse" um outro. Embora o conceito seja esse, algumas publicações relacionadas ao Java e à orientação de objetos fazem abordagens diferentes. Então, podemos considerar basicamente três tipos de polimorfismo: o de métodos, o de classe, o de interface.

4.9.6.1. De métodos

Muitos autores consideram polimorfismo a possibilidade de utilizar dois ou mais métodos com a mesma assinatura, mas com comportamentos (codificação) diferentes. Basta lembrar que já abordamos um recurso da linguagem Java que permite exatamente isso: a sobrescrita. A questão não é avaliar se essa visão está correta ou não. Mesmo porque, de certa forma, a sobrescrita possibilita o uso de um método que pode assumir várias formas (executar tarefas diferentes, de acordo com sua codificação) a partir da mesma chamada, sendo diferenciado pela classe que deu origem ao objeto. Isso justificaria chamar esse recurso de polimórfico, mas acreditamos que é melhor definido como sobrescrita.

4.9.6.2. De classe

Considerando uma hierarquia de classes, temos, em uma superclasse, a generalização de um tipo e, em suas subclasses, a especialização do mesmo tipo. Imagine a seguinte situação: se colocarmos uma cesta à disposição dos clientes da livraria e nela estiver escrito "Coloque aqui seus produtos", estes "produtos" podem ser livros, CDs ou DVDs. Ou ainda, todos eles juntos. Outro exemplo: na livraria, existe uma porta de acesso ao estoque e nela há uma placa com o aviso "Entrada permitida somente para funcionários". Tanto pode entrar um vendedor como um gerente, porque ambos são funcionários.

Esse mesmo princípio se aplica aos programas em Java. Se definirmos um método que recebe um objeto do tipo Produto por parâmetro, podemos passar qualquer objeto instanciado a partir de uma subclasse de Produto que ele será aceito. Da mesma forma, se um método espera um objeto do tipo Funcionario, é possível passar um objeto instanciado a partir da classe Vendedor, por exemplo, que será aceito sem problemas.

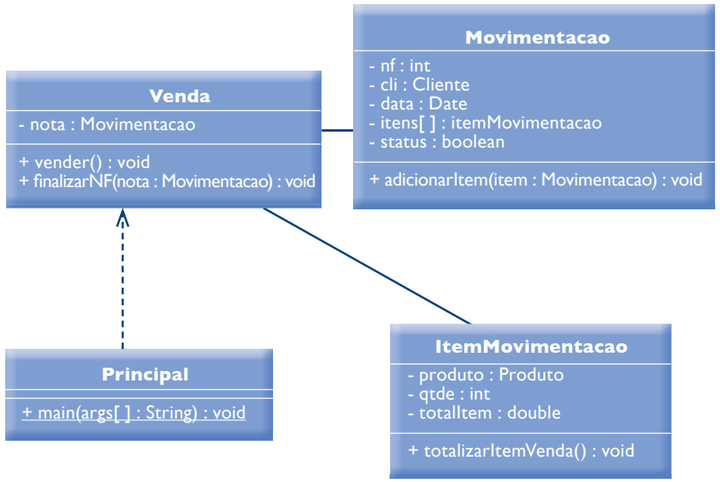
O raciocínio é o seguinte: "Um vendedor é um funcionário", assim como, "Um livro é um produto". A diferença é que vendedor foi definido a partir de uma

especialização da classe Funcionario. Assim, pode ter atributos e métodos específicos referentes a vendedores, porém, não deixa de ser um Funcionario.

Apesar de um livro poder ser utilizado "como se fosse" um Produto, não deixa de "ser" um livro e de ter as características específicas de livro. O polimorfismo, portanto, é a possibilidade de utilizar um objeto como se fosse outro e não transformá-lo em outro. O uso do polimorfismo de classe (ou de tipo) é exemplificado na figura 153, que ilustra o processo de venda da livraria.

Na classe Movimentacao, há um atributo do tipo Cliente, ou seja, nele podemos armazenar um objeto instanciado a partir das classes ClientePF ou ClientePJ. Na classe ItemMovimentacao, temos um atributo do tipo Produto, o que significa poder armazenar nesse atributo um objeto categorizado a partir das classes Livro, CD ou DVD.

Figura 153
Polimorfismo de classe.



4.9.6.3. De interface

Tanto uma superclasse como uma interface podem ser interpretadas como um "superTipo". Ou seja, é possível referenciar um objeto pela interface que sua classe de origem implementa. Voltemos ao exemplo da movimentação da livraria. Para definir quais classes podem ser utilizadas na movimentação de mercadorias, criamos a interface EntidadeMovimentacao e a implementamos nas classes Cliente e Fornecedor (figura 154).

A movimentação, considerando compra e venda, ficará como aparece na figura 155.

Temos uma única classe Movimentacao responsável pelos dados do corpo da compra e da venda. Para armazenar o cliente (no caso de venda) ou o fornecedor (no caso de compra), há o atributo destinatário, do tipo EntidadeMovimentacao, interface que as duas classes implementam.

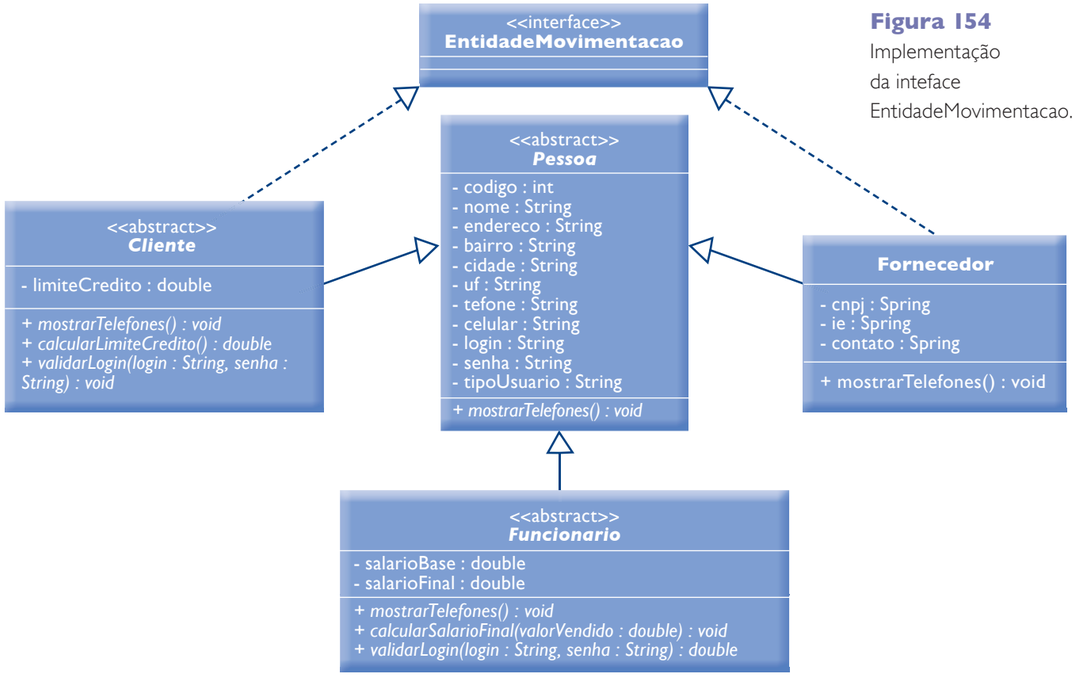


Figura 154
Implementação da inteface EntidadeMovimentacao.

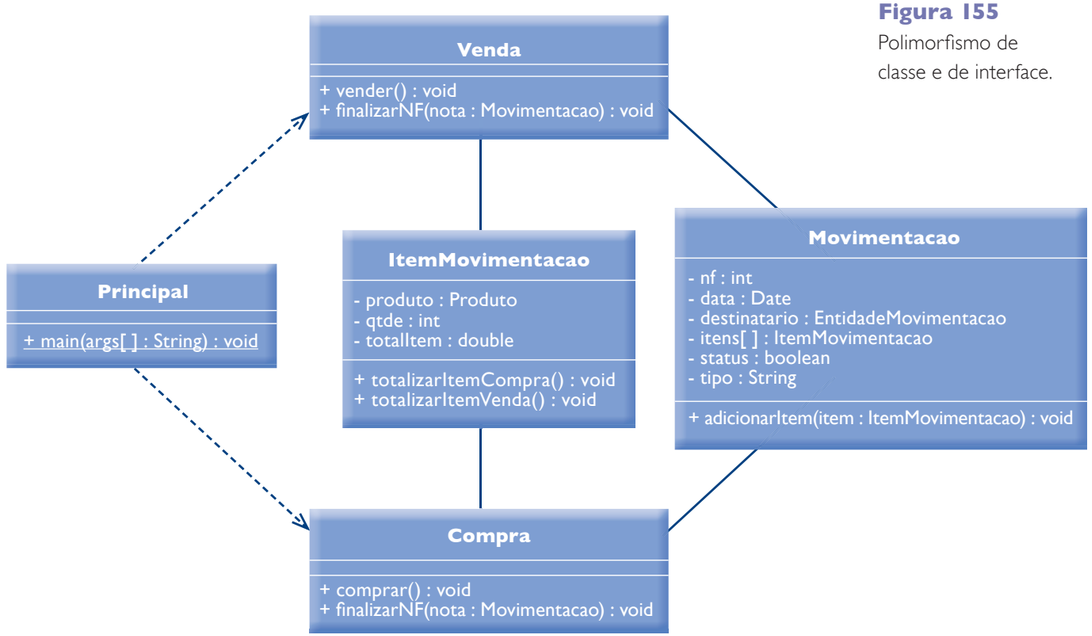


Figura 155
Polimorfismo de classe e de interface.

4.10. Padrões de desenvolvimento de sistemas

Um padrão para desenvolvimento de sistemas estabelece critérios de análise, organização, modelagem, distribuição de tarefas, manutenção, expansão e comunicação entre softwares. Há duas métricas básicas de qualidade que devem ser seguidas por um software: coesão e acoplamento. Um software tem que ter uma alta coesão, isto é, todos os componentes (métodos, classes, pacotes ou qualquer divisão que se faz dependendo do seu tamanho) e devem possuir o mesmo nível de responsabilidade. O ideal é que cada componente esteja focado na resolução de um problema específico. Além disso, precisa haver baixo acoplamento: cada elemento deve ser o mais independente possível de outro. E uma alteração da implementação de um componente não pode afetar nenhum outro.

Os padrões, então, procuram reduzir o acoplamento e aumentar a coesão entre as partes de um sistema, diminuindo, assim, a duplicação do código e possibilitando a consequente reutilização dos componentes. Isso faz com que o custo de manutenção da aplicação caia e a qualidade do código aumente.

4.10.1. Programação em três camadas

Um dos padrões mais utilizados no desenvolvimento de softwares é o que separa, logicamente, uma aplicação em três camadas: **de apresentação, de negócio e de dados**. A vantagem desse tipo de desenvolvimento é que, com a independência das camadas, as atualizações e correções em uma delas podem ser feitas sem prejudicar as demais. Não há, portanto, maior impacto sobre a aplicação como um todo.

A camada de apresentação determina como o usuário interage com a aplicação e como as informações são captadas e apresentadas. A de negócio também é chamada de Lógica empresarial, Regras de negócio ou Funcionalidade. É nela que ficam armazenadas as codificações do funcionamento de todo o negócio. A de dados gerencia a conexão com todas as fontes de dados usadas pelo aplicativo (normalmente, um banco de dados) e o abastecimento de dados dessas fontes para a lógica de negócio.

4.10.2. MVC (Model, View, Controller)

O MVC (iniciais de modelo, visão e controlador) é outro padrão muito utilizado em aplicações orientadas a objetos. Segue o mesmo princípio da separação em camadas, porém, com um foco maior em como esses níveis interagem.

Podemos ver a camada de negócios como o Model (contendo as classes de modelagem da aplicação) e a de apresentação como a View (com toda forma de interação com o usuário: interfaces gráficas, páginas da web etc.). O Controller, porém, é específico dessa arquitetura. É o nível responsável pelas solicitações ao Model (instancia e utiliza os objetos disponíveis) e interação com a View (recebe, manipula e fornece os dados obtidos). A camada de dados, responsável pela comunicação com o banco de dados, não é citada especificamente pela MVC, podendo ser visualizada como pertencente ao Model ou como uma camada independente.

Apesar de existir uma considerável documentação a respeito da organização de softwares em camadas e sobre o MVC, não há uma regra rígida e inflexível de distribuição das classes nessas camadas. Boa parte da decisão fica a critério dos projetistas de sistemas, que adaptam as vantagens dos modelos existentes às necessidades específicas de cada software, podendo, inclusive, criar camadas adicionais como, por exemplo, um nível de persistência de dados. O importante é saber que existem modelos de organização de softwares que agrupam os recursos do sistema por funcionalidade e, em Java, esse agrupamento é feito em packages (pacotes).

4.10.3. Packages

No decorrer do desenvolvimento de um software, muitas classes são criadas e, logo, nos deparamos com a necessidade de organizá-las. Os packages servem para agrupar as classes, normalmente por funcionalidades comuns. Além das vantagens vistas anteriormente, as classes são ordenadas de maneira lógica e física, pois, para cada package, é criado um diretório em disco para salvamento. Como exemplo no projeto Livraria, todas as classes estão contidas no default package, como se observa na figura 156.

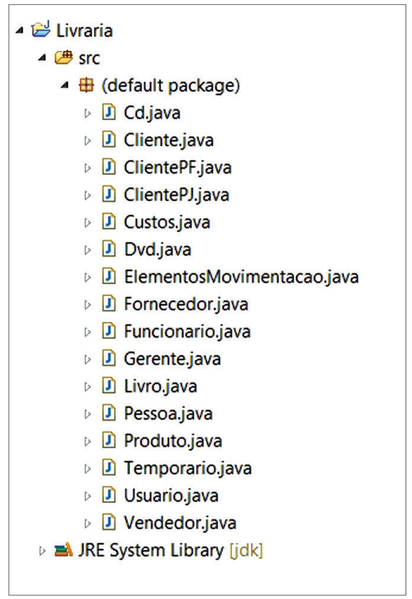


Figura 156
Classes contidas no default package.

O mesmo projeto Livraria devidamente organizado em packages pode ser visualizado nas figuras 157 e 158.

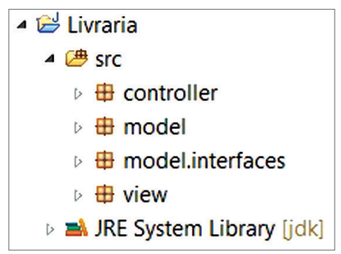


Figura 157
Organização em packages.

Figura 158
Classes organizadas em pacotes (packages).

No eclipse, para criar um novo package, clique no menu File/ New/ Package.

4.11. Interfaces gráficas

A GUI (Graphical User Interface, ou Interface Gráfica com o Usuário), em Java, é um conjunto de classes para disponibilizar componentes gráficos (objetos) como caixas de texto, botões, listas etc. É composta por duas APIs: a AWT e a Swing.

4.11.1. AWT (Abstract Window Toolkit)

A primeira API responsável por interfaces gráficas em Java foi a AWT (Abstract Window Toolkit, ou, literalmente, janela abstrata de caixa de ferramentas), localizada no pacote java.awt, que faz uso dos recursos de cada sistema operacional para desenhar as janelas e os componentes visuais. Essa característica cria uma dependência do sistema em que a aplicação está rodando e impede que o programa tenha o mesmo aspecto visual em diferentes ambientes operacionais. Também reduz os componentes visuais a um conjunto muito pequeno, comum a todos os sistemas operacionais. Por exemplo, um componente visual de árvore de elementos, como no Windows Explorer, não poderia ser implementado em AWT, pois nem todos os sistemas operacionais têm tal componente.

Apesar das limitações, as interfaces gráficas em AWT ainda são utilizadas e muitos de seus componentes permanecem nas atualizações mais recentes, além de ela ser a base para a API Swing. Observe, na figura 159, um exemplo de sua estrutura e alguns de seus componentes, e, na figura 160, veja a classe FormValidacaoUsuario, que dá origem à tela anterior, para depois conferir os principais pontos do código (consulte o quadro *Principais pontos do código*).

Figura 159
Tela de validação de usuários.

Figura 160
Classe FormValidacaoUsuario.

FormularioValidacaoUsuario.java

```
1  import java.awt.*;
2
3  public class FormularioValidacaoUsuario extends Frame{
4
5      protected Dimension dFrame,dLabel,dTextField,dButton;
6      protected Button bValidar, bSair;
7      protected TextField tfLogin, tfSenha;
8      protected Label lLogin, lSenha;
9
10     public FormularioValidacaoUsuario()
11     {
12         Dimension dFrame = new Dimension(260,200);
13         Dimension dLabel = new Dimension(40,25);
14         Dimension dTextField = new Dimension(150,25);
15         Dimension dButton = new Dimension(100,25);
16
17         setTitle("Validação de usuários");
18         setResizable(false);
19         setSize(dFrame);
20         setLocation(400,200);
21         setLayout(null);
22
23         lLogin = new Label("Login:");
24         lLogin.setSize(dLabel);
25         lLogin.setLocation(25,50);
26
27         tfLogin = new TextField();
28         tfLogin.setSize(dTextField);
29         tfLogin.setLocation(75,50);
30
31         lSenha = new Label("Senha:");
32         lSenha.setSize(dLabel);
33         lSenha.setLocation(25,100);
34
35         tfSenha = new TextField();
36         tfSenha.setSize(dTextField);
37         tfSenha.setLocation(75,100);
38
39         bValidar = new Button("Validar usuário");
40         bValidar.setSize(dButton);
41         bValidar.setLocation(25,150);
42
43         bSair = new Button("Sair");
44         bSair.setSize(dButton);
45         bSair.setLocation(130,150);
46
47         add(lLogin);
48         add(tfLogin);
49         add(lSenha);
50         add(tfSenha);
51         add(bValidar);
52         add(bSair);
53     }
54
55 }
```

Principais pontos do código

- Linha 01: importa a API AWT e a disponibiliza para uso.
- Linha 03: o comando extends é responsável por estabelecer a herança entre classes. Assim, a classe FormValidacaoUsuario passa a ser subclasse da classe Frame, que, por sua vez, pertence à API AWT e contém um conjunto de componentes gráficos que podemos utilizar agora, graças à herança.
- Linha 05 a 08: é a declaração dos componentes, necessária, assim como os atributos, para classes que utilizam elementos gráficos. Nesse exemplo, usa-se Button (botões), Label (rótulos), TextField (caixas de texto) e Dimension (um tipo especial, cuja finalidade é definir o tamanho - altura e largura - dos componentes).
- Linha 10: é a assinatura do construtor da classe, ou seja, dentro desse método serão definidos quais componentes irão compor a tela e como.
- Linha 12 a 15: criação dos objetos dimension que contêm as dimensões (altura e largura) usadas em cada componente.
- Linha 17 a 21: alteração de algumas propriedades do Frame (janela) como título (setTitle()); possibilidade de a janela ser ou não redimensionada (setResizable()); tamanho (setSize(), fazendo uso de um objeto dimension (dFrame) para definição desse tamanho; localização em relação ao monitor (setLocation()), e definição de se será utilizado um gerenciador de layouts (setLayout(null)).
- Linha 23 a 25: instancia um objeto do tipo Label e define seu tamanho e localização (em relação ao Frame). Nas linhas seguintes, são instanciados os demais componentes e apresentados seus tamanhos e sua localização.
- Linha 47 a 52: os componentes que foram criados e configurados são adicionados à janela (Frame) para montar a tela.

A classe FormValidacaoUsuario é instanciada e exibida a partir da classe Principal, reproduzida na figura 161.

Principal.java

```
1
2  public class Principal {
3
4      public static void main(String[] args) {
5
6          FormularioValidacaoUsuario form = new FormularioValidacaoUsuario();
7          form.setVisible(true);
8      }
9  }
```

Figura 161
Classe Principal.

Ao ser executada a classe Principal, a tela reproduzida antes, na figura 159, será exibida, mas não executará nenhuma ação, nem mesmo funcionará o botão com um "x" no canto superior direito da janela. É que, para colocar eventos na janela, devemos implementar interfaces específicas para essa finalidade.

4.11.2. Interfaces listeners

Para controlar os eventos de um formulário, devemos incluir na classe as interfaces gerenciadoras de eventos, também chamadas de listeners. Um listener pode ser entendido como um "ouvinte" ou um "rastreador", que consegue capturar um evento ocorrido no formulário e permite vincular a ele, uma ação (codificação). Entre os diferentes tipos de eventos que podem ocorrer, os mais comuns são:

- **Eventos de ação (em componentes):** implementados pela interface ActionListener.
- **Eventos de janela:** implementados pela interface WindowListener.
- **Eventos de teclado:** implementados pela interface KeyListener.
- **Eventos de mouse:** implementados pela interface MouseListener.

Cada um dos listeners possui métodos que capturam os eventos ocorridos. A interface ActionListener possui apenas um, que é:

- actionPerformed().

A interface WindowListener tem sete:

- windowActivated(), windowDeactivated(), windowIconified(), windowDeiconified(), windowOpened(), windowClosed() e windowClosing().

A interface KeyListener conta com três:

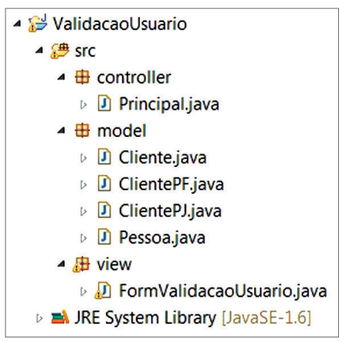
- keyPressed(), keyReleased() e keyTyped().

A interface MouseListener tem cinco:

- mouseCliecked(), mouseEntered(), mouseExited(), mousePressed() e mouseReleased().

A implementação de uma interface é feita pelo comando implements, inserido na abertura da classe. Os métodos das interfaces listeners são abstratos. Portanto, ao implementá-los, todos os seus métodos devem estar inseridos na classe. Vamos, então, implementar as interfaces listeners WindowListener (para ler eventos de janela) e ActionListener (para ler eventos de componentes) na classe FormValidacaoUsuario. Observe, na figura 162, a estrutura (organização de packages) do projeto ValidacaoUsuario.

Figura 162
Projeto ValidacaoUsuario.



Veja como ficou a classe FormValidarUsuario após a atualização (figura 163) e, em seguida, confira uma análise da codificação.

Figura 163
Classe FormValidarUsuario atualizada.

```
FormValidacaoUsuario.java
 1 package view;
 2 import java.awt.*;
 3 import java.awt.event.ActionEvent;
 4 import java.awt.event.ActionListener;
 5 import java.awt.event.WindowEvent;
 6 import java.awt.event.WindowListener;
 7 import javax.swing.JOptionPane;
 8
 9 public class FormValidacaoUsuario
10       extends Frame
11       implements WindowListener, ActionListener {
```

Linha 01: de acordo com a estrutura de packages, é necessário definir a qual classe pertence.

Linha 02 a 07: imports das classes que possuem os recursos para realizar a leitura de eventos.

Linha 09: inclusão das duas listeners na abertura da classe pelo comando implements.

Na próxima codificação, vemos como implementar o WindowListener (figura 164).

Figura 164
Como implementar o WindowListener.

```
78          //Adiciona os objetos ao Frame
79          add(lLogin);
80          add(tfLogin);
81          add(lSenha);
82          add(tfSenha);
83          add(bValidar);
84          add(bSair);
85          //Adiciona o frame no WindowListener
86          addWindowListener(this);
87
88      }
89
90      public void windowActivated(WindowEvent e){
91      }
92
93      public void windowDeactivated(WindowEvent e){
94      }
95
96      public void windowIconified(WindowEvent e){
97      }
98
99      public void windowDeiconified(WindowEvent e){
100     }
101
102     public void windowOpened(WindowEvent e){
103     }
104
105     public void windowClosed(WindowEvent e){
106     }
107
108     public void windowClosing(WindowEvent e){
109         System.exit(0);
110     }
```

Para que o WindowListener funcione, é necessário adicionar o Frame a ele (pelo método addWindowListener(this), visto na linha 86), assim como implementar os sete métodos (abstratos) pertencentes ao WindowListener (linhas 90 a 110), mesmo que só utilizando um deles (windowClosing) para fechar o formulário (linhas 108 a 110).

Vamos ver como é o funcionamento do ActionListener, aproveitando a oportunidade para entender como fazer a interação entre um objeto (originado de uma classe de modelagem, como foi possível observar nos exemplos anteriores) e o formulário, que agora exerce a tarefa de se comunicar com o usuário (camada view). O objeto em questão é do tipo ClientePF e possui, entre outros atributos, um login e uma senha. Esse objeto também conta com um método chamado validarLogin, que recebe duas Strings por parâmetro (login e senha) e as compara com seus atributos (login e senha). Se forem iguais, apresentará uma mensagem "Cliente autorizado!", caso contrário, mostrará "Login desconhecido!". Em seguida, confira a codificação do método validarLogin da classe ClientePF (figura 165).

Figura 165
Codificação do método validarLogin da classe ClientePF.

```
public void validarLogin(String login, String senha) {

    if(this.getLogin().equals(login) &&
       this.getSenha().equals(senha)){

        JOptionPane.showMessageDialog(null,
                "Cliente " + this.getNome() + " autorizado!"
        );

    }else{
        JOptionPane.showMessageDialog(null, "Cliente desconhecido!");
```

Figura 166
Importando o pacote model.

```
1  package view;
2  import java.awt.Button;
3  import java.awt.Dimension;
4  import java.awt.Frame;
5  import java.awt.Label;
6  import java.awt.TextField;
7  import java.awt.event.ActionEvent;
8  import java.awt.event.ActionListener;
9  import java.awt.event.WindowEvent;
10 import java.awt.event.WindowListener;
11
12 import javax.swing.JOptionPane;
13
14 import model.ClientePF;
15
16 public class FormValidacaoUsuario
17        extends Frame
18        implements WindowListener, ActionListener {
19
20    protected Dimension dFrame,dLabel,dTextField,dButton;
21    protected Button bValidar, bSair;
22    protected TextField tfLogin, tfSenha;
23    protected Label lLogin, lSenha;
24
25    ClientePF cliente = new ClientePF(1,"","","","","","",
26                                      "","aluno","123","",
27                                      0,"","",0);
28
29    public FormValidacaoUsuario(){
```

Para fazer a interação do objeto clientePF com a tela de login, instanciamos o objeto na classe FormValidacaoUsuario, inicializando o login com a palavra “aluno” e a senha com “123”, pelo construtor. Mais uma vez, respeitando a organização de packages. Para ter acesso à classe ClientePF, é necessário importar o pacote model (linha 14), conforme ilustra a figura 166.

O próximo passo é adicionar os botões ao ActionListener, pelo método addActionListener(this), para que eles passem a ser monitorados pelo ActionListener (figura 167).

Figura 167
Adicionando botões ao ActionListener.

```
// Define as propriedades do bValidar
bValidar = new Button("Validar usuário");
bValidar.setSize(dButton);
bValidar.setLocation(25,150);
// Adiciona o botão no ActionListener
bValidar.addActionListener(this);

// Define as propriedades do bSair
bSair = new Button("Sair");
bSair.setSize(dButton);
bSair.setLocation(130,150);
// Adiciona o botão no ActionListener
bSair.addActionListener(this);
```

E, por fim, implementar o método actionPerformed (figura 168).

Figura 168
Implementando o actionPermormed.

```
public void actionPerformed(ActionEvent e){

    if (e.getSource() == bValidar){

        if( tfLogin.getText().equals("") ||
            tfSenha.getText().equals("")){

            JOptionPane.showMessageDialog(null,
                    "Favor preencher os dois campos!"
            );

        }else{
            cliente.validarLogin(tfLogin.getText(),tfSenha.getText());
        }

    }

    if (e.getSource() == bSair){
        System.exit(0);
    }
```


Quando um evento for disparado em qualquer componente adicionado ao ActionListener, ele será capturado e o método actionPerformed, executado. Resta, agora, identificar qual componente sofreu o processo. Para isso, utilizaremos o parâmetro do tipo ActionEvent do método actionPerformed e invocaremos seu método getSource(), que retornará o nome do componente que disparou o processo. Se o componente for igual ao bValidar (nome do botão), é verificado se um dos dois textFields estão vazios. Caso não estejam, o método validarLogin do objeto clientePF é acionado, passando por parâmetro o que foi digitado nos textfields tfLogin e tfSenha. Os componentes textFields possuem um método getText() para recuperar o que foi digitado nele e um método setText() que altera o seu valor.

4.11.3. Swing

A API Swing, localizada no pacote javax.swing, é uma atualização da AWT que soluciona - ou pelo menos diminui - a dependência do sistema operacional, característica de interfaces desenvolvidas com AWT. Essa API desenha cada componente ou janela que necessita. Assim, permite que o comportamento visual do programa seja o mesmo em qualquer plataforma, além de oferecer um conjunto extenso de componentes visuais. Para testar as diferenças entre o AWT e a Swing, vamos “transformar” a classe FormValidarUsuario em Swing. Primeiro, vale analisar a abertura da classe (figura 169).

Linha 06 a 11: imports da API Swing.

Linha 15: a herança agora é estabelecida com uma classe JFrame.

Visualmente, a diferença de nomenclaturas de componentes do AWT em relação à Swing é a inserção de um J na frente de cada nome. Além de JFrame, temos a declaração dos componentes utilizados na tela (da linha 19 a 22) todos iniciando com um J.

Figura 169
A classe FormValidacaoUsuario utilizando o swing.

```
1  package view;
2  import java.awt.Button;
3  import java.awt.Dimension;
4  import java.awt.Frame;
5  import java.awt.Label;
6  import java.awt.TextField;
7  import java.awt.event.ActionEvent;
8  import java.awt.event.ActionListener;
9  import java.awt.event.WindowEvent;
10 import java.awt.event.WindowListener;
11
12 import javax.swing.JOptionPane;
13
14 import model.ClientePF;
15
16 public class FormValidacaoUsuario
17        extends Frame
18        implements WindowListener, ActionListener {
19
20    protected Dimension dFrame,dLabel,dTextField,dButton;
21    protected Button bValidar, bSair;
22    protected TextField tfLogin, tfSenha;
23    protected Label lLogin, lSenha;
24
25    ClientePF cliente = new ClientePF(1,"","","","","","",
26                                      "","aluno","123","",
27                                      0,"","",0);
28
29    public FormValidacaoUsuario(){
```

Em relação aos listeners, a implementação explícita não é mais necessária. Eles continuam sendo utilizados, mas, agora, de forma implícita. No trecho seguinte da figura 170, temos o início do construtor e a instanciação dos componentes.

Figura 170
Início do construtor e instanciação dos componentes.

```
24        public FormValidacaoUsuario(){
25
26            super("Validação de usuário");
27            Container tela = getContentPane();
28            setLayout(null);
29
30            lLogin = new JLabel("Login: ");
31            lLogin.setBounds(25,20,50,25);
32
33            tfLogin = new JTextField();
34            tfLogin.setBounds(75,20,150,25);
35
36            lSenha = new JLabel("Senha: ");
37            lSenha.setBounds(25,70,50,25);
38
39            tpSenha = new JPasswordField();
40            tpSenha.setBounds(75,70,150,25);
```

Utilizando a API Swing, o gerenciamento da janela ficou a cargo do componente do tipo Conteiner, que receberá e organizará todos os componentes.

Linha 26: define o título da janela pelo construtor da superclasse (acessado pelo método super()).

Linha 27: cria um componente nomeado como tela do tipo Conteiner.

Linha 28: define que não será utilizado um gerenciador de layout.

Linha 30 a 40: declara os componentes e define tamanho e posicionamento.

O método setBounds() é responsável por definir o tamanho e a localização do componente pelos parâmetros que recebe, na seguinte sequência: coluna, linha, largura e altura. Na figura 171, observe o trecho que define os eventos.

Figura 171
Trecho que define os eventos.

```
42 bOk = new JButton("Ok");
43 bOk.setBounds(25,120,100,25);
44 bOk.addActionListener(
45     new ActionListener(){
46         public void actionPerformed(ActionEvent e){
47
48             String senha = new String(tpSenha.getPassword());
49
50             if( tfLogin.getText().equals("") || senha.equals("")){
51                 JOptionPane.showMessageDialog(null, "Favor preencher os dois campos!");
52             }else{
53
54                 if ( clientePF.validarLogin( tfLogin.getText() , senha ) ){
55                     JOptionPane.showMessageDialog(null, "Cliente autorizado!");
56                 }else{
57                     JOptionPane.showMessageDialog(null, "Login desconhecido!");
58                 }
59             }
60         }
61     }
62 );
```


Linha 44: a inclusão de um componente na interface ActionListener (deixando preparado para receber um evento), assim como a utilização de AWT, é realizada pelo método addActionListener. Porém, com a Swing é criado uma ActionListener (linha 45) dentro do método addActionListener e é definido seu método actionPerformed (linha 46). Com o uso do AWT, tínhamos um único método actionPerformed e, nele, identificávamos o componente que sofreu um evento. Com a Swing, há a definição de um ActionListener e de um método actionPerformed para cada componente que pode sofrer um evento.

Em relação às instruções definidas no evento, vale observar o que vem mais adiante.

Linha 48: para leitura da senha foi utilizado um componente do tipo JPasswordField, que substitui a senha digitada por outros caracteres (asteriscos ou bolinhas, por exemplo). Para recuperar a senha, é necessário "converter" esses caracteres para String, permitindo que sejam utilizados.

Na sequência, é verificado se o login ou a senha estão vazios (linha 50). Então, o método validarLogin do objeto clientePF é invocado, passando-se o login e a senha (lidos pelo formulário) por parâmetros.

Em relação ao exemplo AWT, o método validarLogin retorna true (se os valores passados forem iguais ao seus atributos login e senha) e false, se forem diferentes. A estrutura de desvio condicional if codificada na linha 55 considera esse retorno para apresentar uma das mensagens definidas. A seguir (figura 172), encontra-se a codificação do método validarLogin (já modificado) da classe ClientePF.

```
public boolean validarLogin(String login, String senha) {
    boolean retorno = false;

    if(this.getLogin().equals(login) && this.getSenha().equals(senha)){
        retorno = true;
    }
    return retorno;
}
```

Figura 172
Codificação do método validarLogin já modificado.

A API AWT e a Swing possuem vários componentes que não foram abordados nesses exemplos.

4.12. Tratamentos de exceções

Na execução de um programa, é possível que ocorram **erros de lógica e de execução**, capazes de provocar a sua interrupção, de produzir resultados incorretos ou ainda de causar uma ocorrência inesperada.

Erros de lógica: apresentam-se na elaboração de um algoritmo não apropriado para solucionar o problema. Não causam necessariamente interrupção na execução do programa. Erros de execução: mais específicos se comparados aos lógicos, decorrem de uma operação inválida e causam interrupção na execução do programa, porque este recebe um sinal indicando que a operação não pode ser realizada.

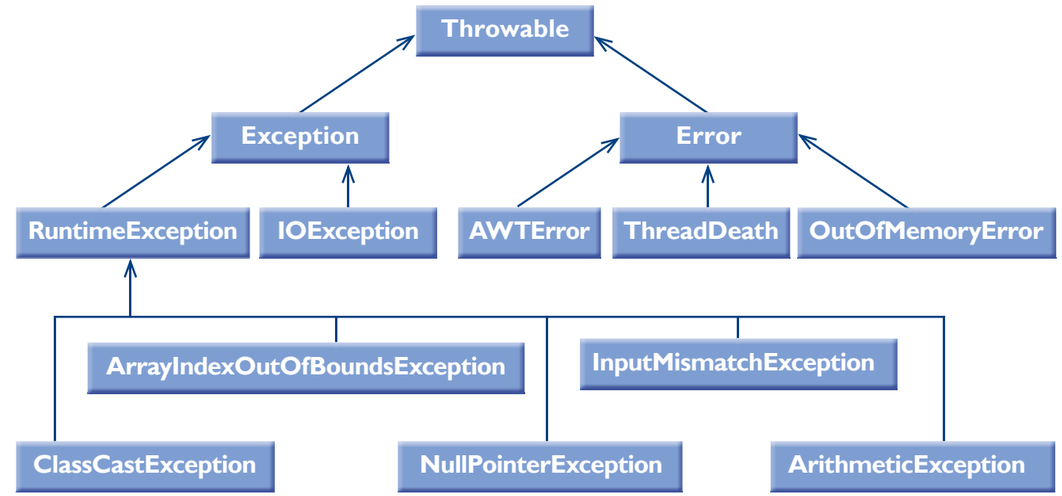


Figura 173
Parte da hierarquia da classe Throwable.

As linguagens de programação possuem formas para identificar e tratar os erros de execução. Em Java, são detectados pela JVM e é criado um objeto de uma classe que caracteriza o erro. O programa que gerou o erro é notificado e, caso seja possível tratá-lo, pode-se acessar o objeto que o caracteriza.

Os erros são caracterizados por objetos de classes específicas que pertencem à hierarquia da classe Throwable. Uma parte dessa hierarquia pode ser visualizada na figura 173.

A classe Throwable é a superclasse da classe Exception e, portanto, também é a superclasse de todas as exceções. Somente objetos Throwable podem ser utilizados como mecanismos de tratamento de exceções. A Throwable tem duas subclasses: Error e Exceptions.

4.12.1. Error

A Error, com suas subclasses, é utilizada para indicar erros graves que não se esperam que sejam tratados pelo programas. Errors raramente acontecem e não são de responsabilidade da aplicação. Exemplos de Errors são os internos da JVM e a falta de memória.

4.12.2. 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. O Java distingue duas categorias de exceções: Unchecked (não verificadas) e Checked (verificadas).

4.12.2.1. Unchecked Exception

Uma exceção não verificada é aquela em que o compilador Java não checa o código para determinar se ela foi capturada ou declarada. Em outras palavras, o programador não é obrigado a inserir o tratamento de erro. De modo geral, pode-se impedir a ocorrência de exceções não verificadas pela codificação adequada. Todos os tipos de exceção, que são subclasses diretas ou indiretas da classe RuntimeException, são exceções não verificadas. São exemplos de Unchecked Exceptions a entrada de tipos incompatíveis (Leitura de uma String em um atributo double, por exemplo); acesso a índice inexistente em um array e chamada a um método de um objeto nulo.

4.12.2.2. Checked Exception

Já em uma Checked Exception, o compilador acusa a possível exceção e obriga o programador a tratá-la. Existem duas formas de tratar uma Checked Exception: usando a cláusula throws ou a estrutura try-catch-finally.

4.12.2.3. Throws

O Throws delega, para o local onde o método foi solicitado, a responsabilidade de tratar o erro. Isso quer dizer que a obrigatoriedade de tratamento é passada para a classe que fará a chamada ao método.

4.12.2.4. 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. Havendo ou não uma exceção (identificada no bloco try) o bloco finally sempre será executado.

Para exemplificar a utilização dos recursos de tratamento de exceções utilizaremos o exemplo de validação de login vistos nos subcapítulos anteriores.

Consideremos que na classe ClientePF o atributo senha é do tipo int e, portanto, para armazenar um valor nele é preciso convertê-lo, já que os componentes textField interpretam qualquer valor lido como String. Observe, na figura 174, a codificação do método validarLogin na classe ClientePF.

```
public boolean validarLogin(String login, int senha) {
    boolean retorno = false;

    if(this.getLogin().equals(login) &&
       this.getSenha() == senha){

        retorno = true;

    }else{

        JOptionPane.showMessageDialog(null,
            "Cliente desconhecido!"
        );
    }

    return retorno;
}
```

Figura 174
Codificação do método validarLogin na classe ClientePF.

Figura 175
Evento do componente bOk na classe FormValidacaoUsuario.

```
42 bOk = new JButton("Ok");
43 bOk.setBounds(25,120,100,25);
44 bOk.addActionListener(
45         new ActionListener(){
46             public void actionPerformed(ActionEvent e){
47
48                 String senhaString = new String(tpSenha.getPassword());
49
50                 int senha = Integer.parseInt(senhaString);
51
52                 if( tfLogin.getText().equals("") || senhaString.equals("")){
53                     JOptionPane.showMessageDialog(null,
54                         "Favor preencher os dois campos!");
55                 }else{
56                     if ( cliente.validarLogin( tfLogin.getText() , senha ) ){
57                         JOptionPane.showMessageDialog(null, "Cliente autorizado!");
58                     }else{
59                         JOptionPane.showMessageDialog(null, "Login desconhecido!");
60                     }
61                 }
62             }
63         }
64 );
```

Na linha 50, a senha obtida por meio do componente tpSenha do tipo JPasswordField precisa ser convertida para int antes de ser passada por parâmetro na chamada do método validarLogin (linha 56). Se for digitada uma letra no campo senha, a conversão para int (linha 50) gerará uma exception do tipo NumberFormatException. Perceba que o compilador não nos obriga a tratar esse erro. Portanto, estamos diante de um Unchecked Exception (outra forma de identificar o tipo da exceção é considerar que NumberFormatException é subclasse de RuntimeException). A inserção da estrutura de try-catch-finally pode tratar essa possível exceção da forma como ilustra a figura 176.

Figura 176
Validação com try-catch-finally.

```
42 bOk = new JButton("Ok");
43 bOk.setBounds(25,120,100,25);
44 bOk.addActionListener(
45         new ActionListener(){
46             public void actionPerformed(ActionEvent e){
47
48                 String senhaString = new String(tpSenha.getPassword());
49
50                 if( tfLogin.getText().equals("") || senhaString.equals("")){
51                     JOptionPane.showMessageDialog(null,
52                         "Favor preencher os dois campos!");
53                 }else{
54
55                     try {
56
57                         int senha = Integer.parseInt(senhaString);
58                         if ( cliente.validarLogin( tfLogin.getText() , senha ) ){
59                             JOptionPane.showMessageDialog(null, "Cliente autorizado!");
60                         }else{
61                             JOptionPane.showMessageDialog(null, "Login desconhecido!");
62                         }
63
64                     } catch (NumberFormatException erro) {
65                         JOptionPane.showMessageDialog(null,
66                                 "Digite apenas números!", "Tipo inválido!", 0
67                         );
68                     }finally{
69                         JOptionPane.showMessageDialog(null,
70                                 "Esta mensagem sempre será exibida!"
71                         );
72                     }
73                 }
74             }
75         }
76 );
```


O bloco de código entre as linhas 55 e 63 pertence ao try. Se alguma exceção ocorrer neste trecho ela será capturada pelo catch da linha 64. Como o possível erro que estamos tratando ocorrerá na conversão de String para int, um objeto do tipo NumberFormatException (a classe de exceção responsável por esse tipo de erro) apreende o erro e é passado por parâmetro para o catch. Dentro do catch (linha 65) é apresentada uma mensagem explicativa para o usuário e o programa continua sua execução. Pelo objeto erro (parâmetro do tipo NumberFormatException utilizado no catch) é possível recuperar mais detalhes da exceção por intermédio de métodos existentes em todas as classes de exceção. Nesse exemplo, o finally foi colocado somente para mostrar que, ocorrendo ou não uma exceção, a mensagem contida nele será exibida.

Podemos também utilizar as exceptions para tratar determinadas situações nos quais pode haver necessidade de realizar verificações que não são necessariamente erros, mas sim valores inválidos, no contexto da aplicação desenvolvida. Por exemplo, consideremos que, para ser válida, a senha (do exemplo de validação de login) deve estar entre 0 e 1000. Essa é uma regra da aplicação, porém, analisando o tipo int no qual a senha é armazenada, a faixa de abrangência vai de -2.147.483,648 a 2.147.483,647, ou seja, estando nessa faixa de valores o compilador não gerará uma exceção. Mas para a minha aplicação, um valor informado fora da faixa entre 0 e 1000 é uma exceção. Para esses casos, podemos utilizar a cláusula throw (não confundir com throws, que tem outro sentido, estudado mais adiante), que realiza uma chamada a uma exceção (força uma exceção). Vejamos, na figura 177, como ficará a validação com o uso do throw.

Na linha 58, é verificado se a senha informada está fora da faixa de valores prevista. Se estiver, a exception IllegalArgumentException é invocada e capturada pelo catch da linha 71, onde uma mensagem será exibida.

Para fechar nossos exemplos de tratamentos de exceção, vamos analisar o comportamento do Java (e os recursos do eclipse) em relação a Checked Exceptions

Figura 177
Validação com uso do throw.

```
42 bOk = new JButton("Ok");
43 bOk.setBounds(25,120,100,25);
44 bOk.addActionListener(
45         new ActionListener(){
46             public void actionPerformed(ActionEvent e){
47
48                 String senhaString = new String(tpSenha.getPassword());
49
50                 if( tfLogin.getText().equals("") || senhaString.equals("")){
51                     JOptionPane.showMessageDialog(null,
52                         "Favor preencher os dois campos!");
53                 }else{
54
55                     try {
56                         int senha = Integer.parseInt(senhaString);
57
58                         if(senha < 0 || senha > 100){
59                             throw new IllegalArgumentException();
60                         }
61
62                         if ( cliente.validarLogin( tfLogin.getText() , senha ) ){
63                             JOptionPane.showMessageDialog(null, "Cliente autorizado!");
64                         }else{
65                             JOptionPane.showMessageDialog(null, "Login desconhecido!");
66                         }
67                     } catch (NumberFormatException erro) {
68                         JOptionPane.showMessageDialog(null,
69                                 "Digite apenas números!", "Tipo inválido!", 0
70                         );
71                     } catch (IllegalArgumentException erro) {
72                         JOptionPane.showMessageDialog(null,
73                                 "A senha informada está fora da faixa permitida!"
74                         );
75                     }
76                 }
77             }
78         }
79 );
```

Figura 178
Observação de erro de uma checked exception.

```
public boolean validarLogin(String login, int senha) {
    boolean retorno = false;

    if(this.getLogin().equals(login) &&
       this.getSenha() == senha){

        new java.io.FileReader("confidencial.txt");

        retorno = true;
    }
    return retorno;
}
```

Figura 179
Opções de tratamento.

```
public boolean validarLogin(String login, int senha) {
    boolean retorno = false;

    if(this.getLogin().equals(login) &&
       this.getSenha() == senha){

        new java.io.FileReader("confidencial.txt");
    }
    retu
}
```

Add throws declaration
Surround with try/catch
Assign statement to new local variable (Ctrl+2, L dir
Assign statement to new field (Ctrl+2, F direct acce

Figura 180
Utilizando o try-catch.

```
public boolean validarLogin(String login, int senha) {
    boolean retorno = false;

    if(this.getLogin().equals(login) &&
       this.getSenha() == senha){

        try {

            new java.io.FileReader("confidencial.txt");

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }

        retorno = true;
    }
    return retorno;
}
```

Figura 181
Uso do throws na classe ClientePF.

```
public boolean validarLogin(String login, int senha)
                            throws FileNotFoundException {
    boolean retorno = false;

    if(this.getLogin().equals(login) &&
            this.getSenha() == senha){

        new java.io.FileReader("confidencial.txt");

        retorno = true;
    }
    return retorno;
}
```


Figura 182
Uso do throws na classe FormValidacaoUsuario.

```
44 bOk = new JButton("Ok");
45 bOk.setBounds(25,120,100,25);
46 bOk.addActionListener(
47         new ActionListener(){
48             public void actionPerformed(ActionEvent e){
49
50                 String senhaString = new String(tpSenha.getPassword());
51
52                 if( tfLogin.getText().equals("") || senhaString.equals("")){
53                     JOptionPane.showMessageDialog(null,
54                         "Favor preencher os dois campos!");
55                 }else{
56
57                     try {
58
59                         int senha = Integer.parseInt(senhaString);
60
61                         if ( cliente.validarLogin( tfLogin.getText() , senha ) ){
62                             JOptionPane.showMessageDialog(null, "Cliente autorizado!");
63                         }else{
64                             JOptionPane.showMessageDialog(null, "Login desconhecido!");
65                         }
66
67                     } catch (NumberFormatException erro) {
68                         JOptionPane.showMessageDialog(null,
69                                 "Digite apenas números!", "Tipo inválido!", 0
70                         );
71                     } catch (FileNotFoundException erro) {
72                         erro.printStackTrace();
73                     }
74                 }
75             }
76         }
77 );
```

e ao uso de throws. Para testarmos as Checked Exceptions, utilizaremos a classe FileReader, responsável por leitura de arquivos em disco. Não entraremos em detalhes sobre essa e outras classes responsáveis por manipulação de arquivos. Por hora, basta saber que, ao usá-la, uma exceção do tipo FileNotFoundException pode ser gerada e essa é uma Checked Exception.

Em nosso exemplo de validação de login, consideremos agora que, ao logar, o usuário terá acesso ao conteúdo de um arquivo texto, nomeado como confidencial.txt, salvo no HD da máquina no qual o programa está sendo executado. Ao digitar a linha de comando que realiza a leitura do arquivo, uma observação de erro aparece no eclipse da forma ilustrada na figura 178.

Parando com o ponteiro do mouse sobre a observação de erro, será exibido o menu suspenso ilustrado na figura 179.

Nesse momento, há duas alternativas (como o próprio eclipse sugere): inserir um try-catch para tratar a exceção no próprio método, ou, utilizar a cláusula throws para postergar o tratamento da exceção, passando essa responsabilidade para a classe que irá realizar a chamada desse método. Inserindo o try-catch, a exceção é tratada, como vemos no próximo código (figura 180). Já o throws é utilizado da forma como sugere a figura 181.

Na assinatura do método, é inserida na cláusula throws e na exception que deverá ser tratada quando o método for utilizado. O compilador para de exigir o tratamento no método e passa a cobrá-lo na sua chamada, em nosso exemplo, na classe FormValidacaoUsuario (figura 182).

Como a linha que faz a chamada ao método validarLogin (linha 61) já está dentro de um bloco de try, basta acrescentar o catch da exception redirecionada pelo throws, no caso, FileNotFounException (linha 71).

É possível capturar uma exceção recorrendo à classe Exception (classe mãe das exceções), ou seja, qualquer erro será interceptado. Esse tipo de tratamento é desaconselhável e deve sempre que possível ser evitado devido à falta de especificação do erro ocorrido. As formas de tratamento de exceções em Java não foram esgotadas nesse capítulo, existindo ainda a possibilidade de criação de nossas próprias classes de exceções e a manipulação de várias outras exceptions.

Exemplos de exceptions mais comuns

- ArithmeticException: resultado de uma operação matemática inválida.
- NullPointerException: tentativa de acessar um objeto ou método antes que seja instanciado.
- ArrayIndexOutOfBoundsException: tentativa de acessar um elemento de um vetor além de sua dimensão (tamanho) original.
- NumberFormatException: incompatibilidade de tipos numéricos.
- FileNotFoundException: arquivo não encontrado.
- ClassCastException: tentativa de conversão incompatível.

4.13. Conexão com banco de dados

O MySQL é um banco de código-fonte aberto, gratuito e está disponível tanto para o Windows como para o Linux. O download pode ser feito diretamente do site do fabricante (http://dev.mysql.com/downloads/). A versão utilizada nos exemplos deste item é a 5.0.

Por definição, um banco de dados é um conjunto de informações armazenadas em um meio físico (papel, disco rígido etc.), organizadas de tal forma que seja possível fazer sua manutenção (inclusão, alteração e exclusão) e diferentes formas de pesquisas. Considerando os bancos de dados informatizados, os SGBDs (Sistemas Gerenciadores de Bancos de Dados) possuem recursos para manutenção, acesso, controle de usuários, segurança, e outras ferramentas de gerenciamento. O SQL (Structured Query Language, ou linguagem de consulta estruturada) é uma linguagem de manipulação de dados que se tornou padrão para SGBDRs (Sistemas Gerenciadores de Bancos de Dados Relacionais). Entre os sistemas de gerenciamento de SGBDRs populares estão: o Microsoft SQL Server, o Oracle, o IBM DB2, o PostgreSQL e o **MySQL**.

4.13.1. JDBC (Java Database Connectivity)

É a API (conjunto de classes e interfaces) responsável pela conexão de programas desenvolvidos em Java com vários tipos de bancos de dados. O próprio JDBC (confira o quadro *Principais funcionalidades do JDBC*) foi desenvolvido em Java e é, portanto, inteiramente compatível com as classes responsáveis pela conexão e manipulação dos dados.

Principais funcionalidades do JDBC

- Estabelecer a conexão com o banco de dados
- Executar comandos SQL
- Permitir a manipulação de resultados (dados) obtidos a partir da execução de comandos SQL
- Gerenciar transações (ações realizadas simultaneamente por um a um dos usuários conectados ao SGBD)
- Capturar e possibilitar o tratamento de exceções relacionadas à conexão e utilização do banco.

Para que um sistema (projeto) Java possa se conectar e gerenciar uma conexão com banco de dados, deve ser configurado para ter acesso ao(s) driver(s) JDBC referente ao(s) banco(s) utilizado(s). A JDBC suporta quatro categorias de drivers: a de tipo 1, que é a ponte JDBC-ODBC; a de tipo 2, API nativa parcialmente Java; a de tipo 3, o Protocolo de rede totalmente Java; e a de tipo 4, o Protocolo nativo totalmente Java. As características de cada um deles estão definidas a seguir.

Tipo 1: JDBC-ODBC bridge driver (Ponte JDBC-ODBC)

O ODBC (Open Data Base Connectivity) é o recurso padrão disponibilizado pela Microsoft, responsável pela conexão a bancos de dados na plataforma Windows. O driver JDBC do Tipo 1 provê a comunicação entre o ODBC e a API JDBC, que é o padrão de conexão a bancos de dados para aplicações desenvolvidas em Java. Sua vantagem é ser ideal para integração de aplicações Java em ambientes que já possuem aplicações desenvolvidas para a plataforma Windows. Há, porém, duas desvantagens: não é indicado para aplicações em grande escala, pois seu desempenho cai à medida que as chamadas JDBC trafegam por meio da ponte para o driver ODBC; e o driver JDBC-ODBC precisa estar instalado no cliente.

Tipo 2: Native API partly Java driver (API nativa parcialmente Java)

Os drivers JDBC do Tipo 2 permitem a comunicação de aplicações Java com os drivers nativos dos SGBDs (geralmente desenvolvidos em C ou C++). Nesse sentido, são semelhantes aos do tipo 1, porém, criam uma ponte entre os drivers nativos de cada SGBD com as aplicações Java. Sua vantagem é ter um desempenho melhor que o driver de Tipo 1. Já a desvantagem é que o driver nativo específico do SGBD utilizado deve estar instalado na máquina cliente, o que impossibilita aplicações web, pois é necessário ter acesso às máquinas clientes.

IMPORTANTE:
A partir desse ponto é preciso que os conceitos relacionados à estrutura dos bancos de dados relacionais e a linguagem SQL já tenham sido estudados e devidamente aprendidos. Caso ainda falte alguma etapa para essa consolidação, vale uma boa consulta às diversas publicações disponíveis sobre o assunto, bem como documentações referentes ao MySQL.

Tipo 3: Net protocol all Java driver (Protocolo de rede totalmente Java)

O driver Tipo 3 converte as requisições do SGBD em um protocolo de rede genérico (não vinculado a nenhum SGBD). Essas requisições são enviadas a um servidor middle-tier, que as traduz e encaminha para um SGBD específico. O middle-tier funciona como camada intermediária, que implementa certas regras de conversão e acesso. Oferece duas vantagens: como o driver e o servidor intermediário são alocados no servidor, não há a necessidade de instalação e configuração das máquinas clientes; é indicado para aplicações web. Há uma só desvantagem: exige a instalação e configuração do servidor intermediário, bem como dos driver nativos dos SGBDs utilizados.

Tipo 4: Native protocol all Java driver (Protocolo nativo totalmente Java)

O driver do Tipo 4 é inteiramente desenvolvido em Java e converte as requisições de um SGBD em um protocolo de rede específico para o SGBD utilizado, assim é realizada uma conexão direta entre a aplicação Java e o driver. Apresenta duas vantagens: por ser totalmente desenvolvido em Java e permitir conexão direta com o SGBD, a aplicação fica independente de plataforma e de instalações na máquina cliente; e também é um drive ideal para aplicações web. A desvantagem é que cada SGBD tem seu próprio driver e nem todos são gratuitos.

Nos exemplos a seguir, o driver usado é o com.mysql.jdbc.Driver (mysql-connector-java-5.1.6-bin), do tipo 4. Os próprios fabricantes dos SGBDs fornecem seus drivers JDBC. Existem muitos fornecedores independentes que desenvolvem e disponibilizam drivers JDBC de vários SGBDs.

4.13.2. Implementação do acesso a dados

Em nosso exemplo do projeto da livraria, uma das manutenções disponíveis é relacionada aos clientes e entre aqueles que constituem pessoa física e os de pessoa jurídica. Utilizaremos a manutenção dos primeiros para exemplificar a conexão e o gerenciamento de dados em um banco MySQL. A classe FormClientePF gera o formulário visualizado na figura 183.

Componentes da API Swing foram utilizados para montar o layout do formulário. Vinculados aos eventos dos botões, estão os códigos que realizam a captura dos dados obtidos pelo formulário e as chamadas aos métodos de um objeto nomeado como clientePF do tipo ClientePF e que será o responsável pela interação com o banco de dados. Na classe ClientePF foram implementados métodos que realizam cada uma das operações com o banco de dados (inclusão, alteração, exclusão, listagem de todos os clientes pessoa física e pesquisa de um determinado cliente).

Detalhemos, então, esse relacionamento entre a obtenção de dados pelo formulário e a chamada aos métodos do objeto clientePF. Primeiramente, na classe ClientePF (no qual será feita a conexão com o banco e o gerenciamento de dados), para termos acesso aos métodos do Java responsáveis pela interação com bancos de dados, deverão ser acrescentados os imports relacionados na figura 184.

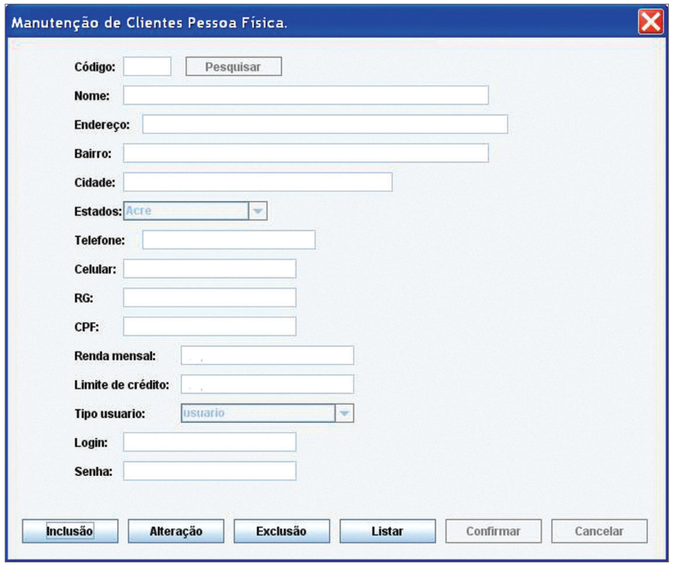


Figura 183
Formulário de manutenção de clientes pessoa física.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

Figura 184
Importações das API de conexão a banco.

Para estabelecer a conexão com o banco, o método responsável (getConnection) precisa de quatro informações: o driver JDBC que será utilizado, o endereço do banco desejado, o login e a senha de identificação no banco de dados. Em função da maior clareza de código, declaramos quatro atributos para receber estas informações (figura 185).

No exemplo, usamos o driver JDBC do MySQL, nosso banco foi criado com o nome livraria, está localizado no caminho padrão do MySQL e não definimos (no banco) nenhum login e senha específicos. Estamos considerando, então, o banco livraria contendo uma tabela chamada clientes.

```
private String servidor = "com.mysql.jdbc.Driver";
private String urlBanco = "jdbc:mysql://localhost/livraria";
private String usuarioBanco = "";
private String senhaBanco = "";
```

Figura 185
Definição de parâmetros de conexão como atributos.

4.13.2.1. Classe java.sql.Statement

A classe java.sql.Statement permite a execução dos comandos fundamentais de SQL. O método Connection.createStatement() é utilizado para criar um objeto do tipo Statement, que representa uma query (um comando SQL). Existem dois métodos de execução de query na classe **Statement**.

1.Statement.executeQuery(): executa uma query e retorna um objeto do tipo java.sql.ResultSet (responsável por armazenar dados advindos de uma consulta SQL) com o resultado obtido (utilizado com select).
2.Statement.executeUpdate(): executa uma query e retorna a quantidade (um valor inteiro) de linhas afetadas (utilizado com insert, update e delete).

4.13.2.2. A classe PreparedStatement

Já a classe java.sql.PreparedStatement contém os métodos da classe Statement e outros recursos para elaboração de query, bem como a possibilidade de passagem de parâmetros.

4.13.2.3. A classe CallableStatement

A classe java.sql.CallableStatement, por sua vez, permite executar procedimentos e funções armazenados no banco como, por exemplo, chamadas a stored procedures.

4.13.3. Inclusão

Na Classe FormClientePF, os objetos clientePF e clientePFAuxiliar (previamente instanciados) são do tipo ClientePF e possuem os atributos referentes aos dados lidos pelo formulário, para a inclusão. Os atributos do objeto clientePFAuxiliar são preenchidos com os valores lidos do formulário como se observa nas linhas 289 a 302, ilustrada na figura 186.

Com os dados já armazenados em seus atributos, o objeto clientePFAuxiliar é passado por parâmetro para o método incluir do objeto clientePF (na linha 304).

Figura 186
Método acaoInclusao na classe FormClientePF.

```
287-public void acaoInclusao() {
288
289     clientePF.setNome( tfNome.getText() );
290     clientePF.setEndereco( tfEndereco.getText() );
291     clientePF.setBairro( tfBairro.getText() );
292     clientePF.setCidade( tfCidade.getText() );
293     clientePF.setUf( cbEstado.getSelectedItem().toString() );
294     clientePF.setTelefone( tfTelefone.getText() );
295     clientePF.setCelular( tfCelular.getText() );
296     clientePF.setLogin( tfLogin.getText() );
297     clientePF.setSenha( tfSenha.getText() );
298     clientePF.setTipoUsuario( cbTipoUsuario.getSelectedItem().toString() );
299     clientePF.setLimiteCredito( Double.parseDouble( ftfLimiteCredito.getText() ) );
300     clientePF.setRg( tfRG.getText() );
301     clientePF.setCpf( tfCPF.getText() );
302     clientePF.setRendaMensal( Double.parseDouble( ftfRendaMensal.getText() ) );
303
304     clientePF.incluir(clientePF);
305 }
```

Na classe ClientePF, o método incluir recebe um parâmetro do tipo ClientePF, também nomeado clientePF (o nome do parâmetro, aliás, poderia ser qualquer um, desde que o tipo seja respeitado) e realiza os comandos que podem ser visualizados em seguida, na figura 187.

Figura 187
Método incluir na classe ClientePF.

```
322-public void incluir(ClientePF clientePF) {
323
324     Connection con = null;
325     PreparedStatement ps = null;
326     try {
327         Class.forName(this.getServidor());
328         con = DriverManager.getConnection(this.getUrlBanco(),
329                                           this.getUsuarioBanco(),
330                                           this.getSenhaBanco());
331         Statement stmt = con.createStatement();
332         String sql = "select max(cliCod) from clientes";
333         ResultSet rs = stmt.executeQuery(sql);
334         rs.next();
335         int proximoCodigo = rs.getInt(1) + 1;
336         rs.close();
337         String sqlInsert = "insert into clientes values(" +
338         "?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ? ,?);";
339         ps = con.prepareStatement(sqlInsert);
340         ps.setInt(1, proximoCodigo);
341         ps.setString(2, "PF");
342         ps.setString(3,clientePF.getNome());
343         ps.setString(4,clientePF.getEndereco());
344         ps.setString(5,clientePF.getBairro());
345         ps.setString(6,clientePF.getCidade());
346         ps.setString(7,clientePF.getUf());
347         ps.setString(8,clientePF.getTelefone());
348         ps.setString(9,clientePF.getCelular());
349         ps.setString(10,clientePF.getLogin());
350         ps.setString(11,clientePF.getSenha());
351         ps.setString(12,clientePF.getTipoUsuario());
352         ps.setDouble(13,clientePF.getLimiteCredito());
353         ps.setString(14,clientePF.getRg());
354         ps.setString(15,clientePF.getCpf());
355         ps.setDouble(16,clientePF.getRendaMensal());
356         ps.executeUpdate();
357
358     } catch (ClassNotFoundException e) {
359        e.printStackTrace();
360     } catch (SQLException e) {
361        e.printStackTrace();
362     } finally{
363        try{
364            ps.close();
365            con.close();
366        }catch(Exception e){
367            e.printStackTrace();
368        }
369     }
370 }
```

A classe Connection é utilizada para armazenar uma conexão com o banco. Na linha 324, um objeto desse tipo é criado e inicializado com null (vazio). Na linha 327, quando o método estático Class.forName() é executado, o driver JDBC do MySQL (contido no atributo servidor) tenta ser inicializado. Carregado o driver, o método Class.forName() irá registrá-lo no java.sql.DriverManager. Na linha 328, a classe DriverManager abre uma conexão com o banco de dados por meio do método getConnection, utilizando informações contidas nos atributos urlBanco, usuarioBanco e senhaBanco. A classe Connection designa um objeto, no caso con, para receber a conexão estabelecida.

Na linha 331, é criado um objeto do tipo Statement (com o nome de stmt) e, na linha 333, é executada uma consulta ao banco para obter o último código cadastrado. Este código, armazenado em um objeto do tipo ResultSet (rs), é incrementado (somado mais 1) na linha 335 e usado como novo

código na linha 340. O método next do objeto rs (ResultSet) serve para posicioná-lo no primeiro registro e para passar para o próximo registro (no caso da linha 334, como só haverá um registro como retorno, ele fica no primeiro). Após a utilização do objeto rs devemos fechá-lo, por meio do método close (linha 336).

Observe que todas a linhas de comando que fazem a conexão e a manipulação dos dados no banco estão em uma estrutura de try e as exceções tratadas nos catch são ClassNotFoundException (exceção gerada se uma das classes não for localizada) e SQLException (se um comando SQL não puder ser executado). No bloco de finally, a conexão é fechada.

Na linha 325 é criado um objeto do tipo PreparedStatement, nomeado como ps e contendo null (vazio) e, na linha 339, é inserido um script de SQL (um insert) contendo a definição de parâmetros (caractere ?). Por meio do método set (do próprio objeto ps) podemos inserir valores nesse comando SQL (insert), passando por parâmetro um índice e o valor que queremos inserir. Por exemplo, na linha 342 está sendo incluído o conteúdo do atributo nome do objeto clientePF, no terceiro parâmetro do objeto ps (podemos entender que o terceiro caractere ? foi substituído pelo nome). Observe que pelo próprio método set do objeto ps, podemos definir o tipo que será inserido (setInt, setString, setDouble etc). Finalmente, quando o comando **SQL** (insert) contido no objeto ps está completo (todos os valores armazenados no objeto clientePF foram passados para ele), o método executeUpdade é executado (linha 356) e os valores são incluídos no banco.

4.13.4. Alteração

Na classe FormClientePF, para a alteração, os atributos do objeto clientePFAuxiliar são preenchidos com os dados do formulário (linhas 309 a 323), como se observa na sequência ilustrada na figura 188.

E com os dados já armazenados em seus atributos, o objeto clientePFAuxiliar é passado por parâmetro para o método alterar do objeto clientePF (na linha 325).

Figura 188: Método acaoAlteracao na classe FormClientePF.

```
307=public void acaoAlteracao() {
308
309     clientePF.setCodigo( Integer.parseInt( tfCodigo.getText() ) );
310     clientePF.setNome( tfNome.getText() );
311     clientePF.setEndereco( tfEndereco.getText() );
312     clientePF.setBairro( tfBairro.getText() );
313     clientePF.setCidade( tfCidade.getText() );
314     clientePF.setUf( cbEstado.getSelectedItem().toString() );
315     clientePF.setTelefone( tfTelefone.getText() );
316     clientePF.setCelular( tfCelular.getText() );
317     clientePF.setLogin( tfLogin.getText() );
318     clientePF.setSenha( tfSenha.getText() );
319     clientePF.setTipoUsuario( cbTipoUsuario.getSelectedItem().toString() );
320     clientePF.setLimiteCredito( Double.parseDouble( ftfLimiteCredito.getText() ) );
321     clientePF.setRg( tfRG.getText() );
322     clientePF.setCpf( tfCPF.getText() );
323     clientePF.setRendaMensal( Double.parseDouble( ftfRendaMensal.getText() ) );
324
325     clientePF.alterar(clientePF);
326 }
```

Na classe ClientePF, o método alterar recebe um parâmetro do tipo ClientePF e realiza comandos visualizados na figura 189.

As instruções de conexão e preparação do comando SQL são as mesmas utilizadas e descritas no método incluir. A diferença é que agora o comando SQL executado é o update, substituindo os valores contidos no banco

Figura 189
Método alterar na classe ClientePF.

```
375=public void alterar(ClientePF clientePF) {
376
377     Connection con = null;
378     PreparedStatement ps = null;
379
380     try {
381         Class.forName(this.getServidor());
382         con = DriverManager.getConnection(this.getUrlBanco(),
383                                           this.getUsuarioBanco(),
384                                           this.getSenhaBanco());
385
386         String sqlUpdate = "update clientes set cliNome = ?, " +
387                                                "cliEndereco = ?, " +
388                                                "cliBairro = ?, " +
389                                                "cliCidade = ?, " +
390                                                "cliUF = ?, " +
391                                                "cliTelefone = ?, " +
392                                                "cliCelular = ?, " +
393                                                "cliLogin = ?, " +
394                                                "cliSenha = ?, " +
395                                                "cliTipoUsuario = ?, " +
396                                                "cliLimiteCredito = ?, " +
397                                                "cliRG = ?, " +
398                                                "cliCPF = ?, " +
399                                                "cliRendaMensal = ? " +
400                                                "where cliCod = ?;";
401         ps = con.prepareStatement(sqlUpdate);
402         ps.setString(1,clientePF.getNome());
403         ps.setString(2,clientePF.getEndereco());
404         ps.setString(3,clientePF.getBairro());
405         ps.setString(4,clientePF.getCidade());
406         ps.setString(5,clientePF.getUf());
407         ps.setString(6,clientePF.getTelefone());
408         ps.setString(7,clientePF.getCelular());
409         ps.setString(8,clientePF.getLogin());
410         ps.setString(9,clientePF.getSenha());
411         ps.setString(10,clientePF.getTipoUsuario());
412         ps.setDouble(11,clientePF.getLimiteCredito());
413         ps.setString(12,clientePF.getRg());
414         ps.setString(13,clientePF.getCpf());
415         ps.setDouble(14,clientePF.getRendaMensal());
416         ps.setInt(15, clientePF.getCodigo());
417         ps.executeUpdate();
418
419     } catch (ClassNotFoundException e) {
420         e.printStackTrace();
421     } catch (SQLException e) {
422         e.printStackTrace();
423     } finally{
424         try{
425             ps.close();
426             con.close();
427         }catch(Exception e){
428             e.printStackTrace();
429         }
430     }
431 }
```

pelos valores armazenados no objeto clientePF. Essa substituição de valores (alteração) será aplicada somente ao registro cujo campo cliCod é igual ao valor contido no atributo código do objeto clientePF. Esse critério foi definido por intermédio da cláusula where do comando update.

4.13.5. Exclusão

Na classe FormClientePF é necessário obter somente o código do cliente a ser excluído. Tal código é armazenado no objeto clientePFAuxiliar, que é passado por parâmetro para o método excluir do objeto clientePF, como se pode visualizar na figura 190.

Figura 190
Método acaoExclusao na classe FormClientePF.

```
328=public void acaoExclusao() {
329
330     clientePF.setCodigo(Integer.parseInt(tfCodigo.getText()));
331     clientePF.excluir(clientePF);
332 }
```

Na classe ClientePF, o método excluir recebe um objeto do tipo ClientePF como parâmetro e realiza os comandos visualizados na figura 191.

Figura 191
Método excluir na classe ClientePF.

```
427⊖public void excluir(ClientePF clientePF) {
428
429     Connection con = null;
430     PreparedStatement ps = null;
431
432     try {
433         Class.forName(this.getServidor());
434         con = DriverManager.getConnection(this.getUrlBanco(),
435                                           this.getUsuarioBanco(),
436                                           this.getSenhaBanco());
437
438         String sqlDelete = "delete from clientes where cliCod = ?;";
439         ps = con.prepareStatement(sqlDelete);
440         ps.setInt(1, clientePF.getCodigo());
441         ps.executeUpdate();
442
443     } catch (ClassNotFoundException e) {
444         e.printStackTrace();
445     } catch (SQLException e) {
446         e.printStackTrace();
447     } finally{
448
449         try{
450             ps.close();
451             con.close();
452
453         }catch(Exception e){
454             e.printStackTrace();
455         }
456     }
457 }
```

As instruções de conexão e preparação do comando SQL são as mesmas utilizadas e descritas no método incluir. A diferença é que agora o comando SQL executado é o delete, que removerá o registro do banco cujo campo cliCod seja igual ao conteúdo do atributo código do objeto clientePF. Esse critério foi definido por meio da cláusula where do comando delete.

4.13.6. Listagem geral

Na classe FormClientePF, é realizada a chamada ao método listar do objeto clientePF, como se pode ver na figura 192.

Figura 192
Método acaoListar na classe FormClientePF.

```
334⊖public void acaoListar() {
335     clientePF.listar();
336 }
```

Na classe ClientePF, é executada uma consulta ao banco e o retorno são os campos cliCod, cliNome, cliTelefone e cliCelular de todos os registros da tabela clientes (figura 193).

Nesse exemplo, montaremos uma String com o retorno da consulta e a apresentaremos por meio de um showMessageDialog. Um objeto rs tem o formato de uma matriz, em que o retorno de uma consulta pode ser tratado como uma tabela. Os métodos de acesso aos dados contidos em um objeto rs são gets específicos para o tipo de dado armazenado. Identifica-se o campo desejado pelo seu referente índice (coluna) na linha acessada. Na

Figura 193
Método listar na classe ClientePF.

```
194⊖public void listar(){
195
196     Connection con = null;
197     try {
198         Class.forName(this.getServidor());
199         con = DriverManager.getConnection(this.getUrlBanco(),
200                                           this.getUsuarioBanco(),
201                                           this.getSenhaBanco());
202
203         Statement stmt = con.createStatement();
204         String sql = "select cliCod, cliNome, cliTelefone, cliCelular " +
205                      "from clientes where cliTipo = 'PF'";
206
207         ResultSet rs = stmt.executeQuery(sql);
208
209         String relacao = "Relação de clientes pessoa física cadastrados \n";
210         while (rs.next()) {
211             relacao = relacao + "\n Cod: " + rs.getString(1).toString() +
212                                 " - Nome: " + rs.getString(2) +
213                                 " - Telefone: " + rs.getString(3) +
214                                 " - Celular: " + rs.getString(4);
215         }
216         rs.close();
217         JOptionPane.showMessageDialog(null, relacao + "\n");
218
219     } catch (ClassNotFoundException e) {
220         e.printStackTrace();
221     } catch (SQLException e) {
222         e.printStackTrace();
223     } finally{
224
225         try{
226             con.close();
227
228         }catch(Exception e){
229             e.printStackTrace();
230         }
231     }
232 }
```

207 é realizada a consulta. No looping da linha 210 a 215 (while), o rs é percorrido a partir de sua primeira linha (posicionado pelo método next do rs) e passando linha a linha (por meio do mesmo método next) até o fim do rs (último cliente retornado). A cada linha, o conteúdo de cada coluna é concatenado (unido) com trechos de texto que identificam uma a uma das informações (linhas 211 e 214). Observe que rs.getString(1) equivale ao código que por sua vez é do tipo int, mas, como a intenção é montar uma String de mensagem, o valor pode ser recuperado já convertido por meio do método rs.getString(1).toString() (linha 211). Na linha 217, a variável do tipo String, nomeada como relacao (declarada na linha 209), é "montada" por intermédio de concatenações com o conteúdo do objeto.

4.13.7. Pesquisa

Uma pesquisa pode ser realizada de formas diferentes. O que apresentamos, então, é simplesmente um exemplo. Na classe FormClientePF, utilizaremos mais um objeto do ClientePF como auxiliar na pesquisa, criado na linha 340 com o nome de clientePFPesquisa. O objeto clientePF é instanciado novamente dentro do método somente para garantir que estará vazio. A manipulação desses dois objetos é ilustrada na figura 194.

Figura 194
Método acaoPesquisar na classe FormClientePF.

```
338⊖public void acaoPesquisar() {
339
340     ClientePF auxClientePF = new ClientePF();
341     ClientePF clientePF = new ClientePF();
342
343     clientePF.setCodigo(Integer.parseInt(tfCodigo.getText()));
344     auxClientePF = clientePF.pesquisar(clientePF);
345
346     if(auxClientePF.getNome().equals("")){
347         limpaCampos();
348
349     }else{
350
351         tfNome.setText( auxClientePF.getNome() );
352         tfEndereco.setText( auxClientePF.getEndereco() );
353         tfBairro.setText( auxClientePF.getBairro() );
354         tfCidade.setText( auxClientePF.getCidade() );
355         cbEstado.setSelectedItem( auxClientePF.getUf() );
356         tfTelefone.setText( auxClientePF.getTelefone() );
357         tfCelular.setText( auxClientePF.getCelular() );
358         tfLogin.setText( auxClientePF.getLogin() );
359         tfSenha.setText( auxClientePF.getSenha() );
360         cbTipoUsuario.setSelectedItem( auxClientePF.getTipoUsuario() );
361         ftfLimiteCredito.setText( Double.toString( auxClientePF.getLimiteCredito() ) );
362         tfRG.setText( auxClientePF.getRg() );
363         tfCPF.setText( auxClientePF.getCpf() );
364         ftfRendaMensal.setText( Double.toString( auxClientePF.getRendaMensal() ) );
365     }
366 }
```

Na linha 343, o código do cliente a ser pesquisado é armazenado no objeto clientePF. Na 344, esse objeto (clientePF) é passado por parâmetro para o método pesquisar contido nele mesmo!? (é, isso é possível). Como o objeto clientePF foi instanciado novamente dentro desse método, ele contém somente o atributo código preenchido. O método pesquisar procura um registro no banco de dados que tenha o mesmo código informado. Se encontrar, retorna um objeto preenchido com os demais dados, caso contrário retorna um objeto como foi passado por parâmetro, ou seja, somente com o código preenchido.

O retorno da pesquisa é armazenado no objeto clientePFPesquisa (ainda na linha 344), e seu atributo nome é testado na linha 346. Imaginando que nome é um dado obrigatório e que não existirá um cliente cadastrado sem nome, se o atributo nome de clientePFPesquisa estiver em branco é porque o cliente não foi encontrado e, então, é executado um método que limpará os campos do formulário, senão os valores contidos em clientePFPesquisa são carregados no formulário.

Na classe ClientePF, o método pesquisar recebe um objeto do tipo ClientePF como parâmetro, retorna um objeto do tipo ClientePF e realiza a pesquisa da forma como ilustra a figura 195.

Na linha 242 é montado o comando SQL (select) que fará a consulta. Esse comando leva em consideração o conteúdo do atributo código do objeto clientePF passado por parâmetro, e é executado na linha 246. Como a consulta é feita por intermédio da chave primária da tabela clientes (código), somente um registro será retornado (caso seja encontrado). O método next posiciona o rs em seu primeiro registro (que em nosso exemplo, também é o último, já que só teremos um registro de retorno). O método isLast retorna true se o rs estiver posicionado no último registro (e false, se não estiver). Se o rs estiver vazio,

Figura 195
Método pesquisar na classe ClientePF.

```
232⊖public ClientePF pesquisar(ClientePF clientePF){
233
234     Connection con = null;
235     try {
236         Class.forName(this.getServidor());
237         con = DriverManager.getConnection(this.getUrlBanco(),
238                                           this.getUsuarioBanco(),
239                                           this.getSenhaBanco());
240
241         Statement stmt = con.createStatement();
242         String sql = "select * from clientes " +
243                      "where cliTipo = 'PF' and " +
244                      "cliCod = " + clientePF.getCodigo();
245
246         ResultSet rs = stmt.executeQuery(sql);
247         rs.next();
248
249         if( rs.isLast() == false){
250             JOptionPane.showMessageDialog(null, "Cliente não encontrado!");
251         }else{
252
253             clientePF.setCodigo(rs.getInt(1));
254             clientePF.setNome(rs.getString(3));
255             clientePF.setEndereco(rs.getString(4));
256             clientePF.setBairro(rs.getString(5));
257             clientePF.setCidade(rs.getString(6));
258             clientePF.setUf(rs.getString(7));
259             clientePF.setTelefone(rs.getString(8));
260             clientePF.setCelular(rs.getString(9));
261             clientePF.setLogin(rs.getString(10));
262             clientePF.setSenha(rs.getString(11));
263             clientePF.setTipoUsuario(rs.getString(12));
264             clientePF.setLimiteCredito(rs.getDouble(13));
265             clientePF.setRg(rs.getString(14));
266             clientePF.setCpf(rs.getString(15));
267             clientePF.setRendaMensal(rs.getDouble(16));
268         }
269         rs.close();
270
271     } catch (ClassNotFoundException e) {
272         e.printStackTrace();
273     } catch (SQLException e) {
274         e.printStackTrace();
275         JOptionPane.showMessageDialog(null, "Ocorreu um erro na pesquisa!");
276     } catch (NumberFormatException e) {
277         e.printStackTrace();
278         JOptionPane.showMessageDialog(null, "Digite somente números!");
279     } finally{
280         try{
281             con.close();
282         }catch(Exception e){
283             e.printStackTrace();
284         }
285     }
286     return clientePF;
287 }
```

aparecerá antes do fim, portanto, se o método isLast retorna false é porque o rs está vazio. Na linha 249, é testado o método isLast. Se for false, apresenta uma mensagem, senão, armazena os dados do cliente encontrado no objeto clientePF (linha 253 a 267). Na linha 286, o objeto é retornado com ou sem dados dependendo do resultado da consulta.

Esse capítulo não pretende esgotar os recursos e possibilidades de uso da linguagem Java, tampouco das técnicas de programação orientada a objeto. O intuito foi passar pelas principais características dessa linguagem, suas estruturas e organizações, além de demonstrar as aplicações práticas dos principais conceitos da orientação a objetos, que são perfeitamente viáveis nessa linguagem. Ainda há muito o que ser visto tanto sobre as técnicas de orientação a objeto, quanto sobre Java. Esperamos que esse seja o seu ponto de partida.