Programação Java

Senai Informatica - 1.32

Introdução à Tecnologia Java

Breve Histórico

Em 1991, um grupo de funcionários da Sun Microsystems iniciou um projeto da empresa visando o desenvolvimento de programas para pequenos dispositivos eletrônicos de consumo, tais como: o PDA (Personal Green e James Assistant), eletrodomésticos em geral e outros. Ele recebeu o nome de Projeto Green e James Fosling assumiu sua coordenação.

A idéia inicial era realizar a programação dos chips desses dispositivos de modo a aumentar suas possibilidades de uso. Mas o desenvolvimento de programas específicos para cada tipo de equipamento inviabilizaria o projeto. Então, a equipe voltou-se para a construção de um sistema operacional que permitisse a utilização de seus programas por diferentes tipos de equipamentos.

Para desenvolver o novo sistema operacional, aproveitando a experiência do grupo, houve a tentativa de utilizar o C++ como linguagem de programação. No entanto, problemas enfrentados com o uso dessa linguagem levaram James Gosling a mudar a estratégia do projeto: abandonar o C++ e construir uma nova linguagem.

A nova linguagem foi construída e batizada inicialmente com Oak (carvalho), uma referência à árvore que James Gosling visualizava a partir de seu escritório. A arquitetura e desenho dessa linguagem sofreram influência de diversas outras linguagens (como Eiffel, SmallTalk e Objetive C) e sua sintaxe baseou-se nas linguagens C e C++.

O sistema operacional que fora desenvolvido posteriormente pela equipe do Projeto Green foi batizado como GreenOS e junto com ele foi construída uma interface gráfica padronizada.

Tendo uma linguagem de programação adequada, um novo sistema operacional e uma interface gráfica padrão, a equipe de James Gosling passou a dedicar-se ao desenvolvimento do Star7, um avançado PDA. Em 1993, logo após sua conclusão, a Sun Microsystems participou de uma concorrência pública para desenvolvimento de uma tecnologia para TV a cabo interativa (onde seria aplicado o Star7), mas foi vencida.

Não encontrando mercado para o Star7 e tampouco vendo horizontes para outros dispositivos eletrônicos, o Projeto Green estava prestes a ter seu financiamento cortado e seu pessoal transferido para outros projetos. Foi quando a Sun decidiu abandonar a ênfase nos dispositivos eletrônicos e voltar-se para a Internet, que já começava a crescer.

O nome da linguagem desenvolvida pelo Projeto Green foi mudado de Oak para Java, uma vez que já havia outra linguagem com aquele nome. Java é o nome da cidade de origem de um tipo de café importado pelos norte-americanos e consumido pela equipe de James Gosling.

Até 1994 não havia uma aplicação definida para a tecnologia Java. Foi quando Jonathan Payne e Patrick Naughton criaram um novo navegador para a Web (batizado como WebRunner) que podia executar programas escritos em Java via Internet. Esse navegador foi apresentado pela Sun no SunWorld'95 como o navegador HotJava, juntamente com o ambiente de desenvolvimento Java.

Em 1995, A Netscape licenciou a tecnologia Java e lançou uma nova versão de seu navegador Web, que também dava suporte à execução de pequenos aplicativos escritos em Java, denominados applets. Nesse mesmo ano, outras empresas de navegadores Web também lançaram novar versões de seus produtos para dar suporte a Java.

Em 1996, numa iniciativa inédita, a Sun resolveu disponibilizar gratuitamente um kit de desenvolvimento de software para a comunidade, que ficou conhecido como Java Developer's Kit (JDK). Inicialmente, esse conjunto de ferramentas poderia ser utilizado nos sistemas operacionais Solaris, Windows 95 e Windows NT. Posteriormente, foram disponibilizados kits para desenvolvimento em outros sistemas operacionais (como o OS/2, o Linux e o Macintosh).

Desde então, a aceitação da tecnologia Java Cresceu rapidamente entre empresas e desenvolvedores. Em 1997, a Sun lançou o JDK 1.1, com melhorias significativas para o desenvolvimento de aplicações gráficas e distribuídas. Em 1999, ela lançou o JDK 1.2 com outras inovações importantes. Depois disso, a empresa continuou liberando gratuitamente novas versões de sua tecnologia e retomou os investimentos no projeto original de programação para pequenos dispositivos eletrônicos.

Java como tecnologia de Desenvolvimento de Software

Muitas vezes, tem-se utilizado a expressão "linguagem Java" de forma incoerente na mídia e entre os profissionais e acadêmicos, como se ela representasse, de forma completa, a essência do que é Java: uma linguagem. É importante resolver preventivamente alguns possíveis conflitos conceituais relativos ao termo "Java" para que se entenda melhor o que ele representa, de que se compõe.

Enquanto tecnologia de desenvolvimento de software, Java compõe-se de três partes distintas: um ambiente de desenvolvimento, uma linguagem de programação e uma interface de programas aplicativos (Application Programing Interface – API).

O ambiente de desenvolvimento é o conjunto de ferramentas utilizadas para a construção de aplicativos. Faz parte do kit de desenvolvimento padrão de Java um conjunto considerável de ferramentas, tais como: um compilador (javac), um interpretador de aplicativos (java) e um gerador de documentação (javadoc).

Também existem várias IDEs (Integrated Development Environment – Ambiente de Desenvolvimento Integrado) disponíveis para Java e que podem ser agregados ao seu ambiente principal para facilitar o processo de desenvolvimento de software. São exemplos de IDEs: Eclipse, IntelliJ IDEA e NetBeans.

A linguagem Java, por sua vez, é composta por um conjunto de palavras e símbolos utilizados pelos programadores para escrever os programas. Ela é formada por um conjunto de palavras reservadas utilizadas para escrever expressões, instruções, estruturas de decisão, estruturas de controle, métodos, classes e outros.

Mas os programas Java não resultam tão- somente da junção de um ambiente de desenvolvimento e de uma linguagem de programação. A isso soma-se um extenso conjunto de classes e interfaces, que formam sua API.

O termo interface tem um sentido particular em Java e não deve ser confundido com "interface gráfica de usuário". Do mesmo modo que uma classe, uma interface pode abrigar atributos e métodos. Para evitar mal-entendidos, as referências à "interface gráfica de usuário" serão feitas através da sigla GUI (Graphic User Interface – Interface Gráfica de Usuário).

Os programas Java são compostos por classes e interfaces e estas são formadas por atributos e métodos. É possível criar cada classe e interface necessária para a construção de determinado aplicativo. No entanto, isso não faz sentido. Um número enorme de tarefas pode ser realizado utilizando-se classes e interfaces já existentes na API Java, liberando o desenvolvedor para concentrar-se apenas em aspectos particulares da aplicação.

Sendo assim, pode-se dizer que existem três partes distintas para se aprender acerca de Java. A primeira diz respeito ao funcionamento do seu ambiente de desenvolvimento, a segunda é a linguagem de programação e a terceira é a sua extensa biblioteca de classes e interfaces.

Java é, pois, a junção de uma linguagem, um ambiente e uma API, e será tratada nesse estudo como uma Tecnologia.

Java como plataforma

Uma plataforma é um conjunto de elementos que possibilitam a execução de softwares aplicativos. Basicamente, o que você precisa para rodar um aplicativo é um computador e um sistema operacional instalado nele. Mas os sistemas operacionais são concebidos para determinadas arquiteturas de computadores e são incompatíveis com todas as demais. Por isso, os próprios sistemas operacionais costumam ser utilizados como identificadores das plataformas.

Do mesmo modo que os sistemas operacionais são compatíveis apenas com determinado tipo de computador, os programas compilados com as tecnologias tradicionais somente são compatíveis com um sistema operacional. Esse é um problema que tira o sono dos desenvolvedores de software. Se m aplicativo é escrito e compilado com C++ para ser executado no Windows, por exemplo, então não será possível executa- lo em nenhuma outra plataforma (Linux, Solaris, Macintosh, etc.)

Mas a tecnologia Java mudou isso. O mesmo arquivo gerado pelo seu compilador pode ser executado em qualquer sistema operacional e, por conseguinte, em qualquer arquitetura de computador. Isso significa que você escreve e compila seus programas em Java e pode executá-los em qualquer plataforma que Java é suportado.

Em um ambiente Java típico, o programa passa por 5 fases distintas:

1. É criado em um editor e armazenado em disco.

2. É compilado, gerando arquivos com código intermediário (byte-codes).

3. Um verificador de byte-codes verifica se todas as instruções são válidas e se não violam restrições de segurança

4. O interpretador lê os bytecodes e os converte para código binário específico para a plataforma atual.

Note que o processo de compilação de Java gera uma representação intermediária (byte-code) que poderá ser interpretada em qualquer sistema operacional que contenha JRE (Java Runtime Environment – Ambiente de Execução Java). É isso que garante aos aplicativos desenvolvidos com Java a independência de plataforma.

Finalidade

São três tipos básicos de softwares que podem ser desenvolvidos utilizando a tecnologia Java, quais sejam: aplicativos Desktops, aplicativos Web e aplicativos Móveis.

Os aplicativos Desktop caracterizam-se como programas executados pelo usuário sobre o seu sistema operacional e sem a intermediação de um browser (navegador Web). Portanto, é um programa que tende a ser executado em um ambiente desktop (monousuário) ou em uma LAN (Local Area NetWork – Rede Local).

Os aplicativos Web também são programas armazenados em um servidor na Internet. A diferença é que os servlets não são descarregados no computador cliente e sim executados no próprio computador servidor. Quando uma página HTML solicita um serviço a um servlet, o computador servidor onde ele se encontra irá executa-lo e enviar seu retorno para o computador cliente. Esse retorno pode ser muitas coisas diferentes, desde uma nova página HTML estática até um conjunto de informações recuperadas através de consulta a um banco de dados mantido no servidor.

Também é possível incluir programação Java em páginas HTML de um servidor Web utilizando-a como linguagem de script. O JSP (Java Server Pages) e o JFS (Java Server Faces) oferecem a possibilidade de se escrever instruções Java dentro de tags nas próprias páginas HTML. Geralmente, você poderá realizar uma tarefa com eficiência similar utilizando JSP/JFS e servlets.

Os aplicativos Móveis são programas armazenados no Google Play e carregados no aparelho celular Android do cliente quando ele acessa a loja na Internet. Um aplicativo Móvel é executado no celular do usuário permitindo a utilização de vários recursos somente disponíveis nestes aparelhos, tais como GPS, Câmera, Altímetros, etc.

Vantagens

A tecnologia Java é simples, orientada a objetos, segura, portável, compilada, interpretada, independente de plataforma, portável, com tipagem forte e que oferece suporte a programação concorrente e de sistemas distribuídos.

A simplicidade é uma das características mais importantes de Java. É isso que possibilita que a sua aprendizagem possa ocorrer sem a necessidade de treinamentos intensos ou larga experiência anterior. Programadores com conhecimento das linguagens C e C++ encontrarão muitas semelhanças em Java e as assimilarão de forma mais rápida. Além disso, Java é muito mais limpa que C ou C++.

Java é orientada a objetos e, com exceção dos tipos primitivos, tudo é representado na forma de objetos. Até mesmo os tipos primitivos podem ser encapsulados em objetos quando isso for necessário. Os programas são compostos por classes, que representam categorias de objetos e podem herdar atributos e métodos de outras classes.

A ausência de herança múltipla é compensada com uma solução muito melhor: o uso de interfaces. Uma classe pode herdar características de uma super-classe e ainda implementar métodos definidos por uma ou mais interfaces.

Não existem variáveis globais ou funções independentes em Java. Toda variável ou método pertence a uma classe ou objeto e só pode ser invocada através dessa classe ou objeto. Isso reforça o forte caráter orientado a objetos de Java.

Java garante a escrita de programas confiáveis. O processo de compilação elimina uma gama enorme de possíveis problemas e uma checagem dinâmica (realizada em tempo de execução) contorna muitas situações que poderiam gerar erros.

Java elimina vários tipos de erros que geralmente ocorrem em programas escritos em C/C++. Programas escritos com Java não corrompem a memória. No entanto, programas escritos em C/C++ podem alterara qualquer posição da memória do computador e, por isso, corromper dados e comprometer a sua execução normal.

Em programas escritos em Java, um sistema automático de gerenciamento de memória realiza a tarefa de liberar recursos. Um processo chamado de coletor de lixo (garbage collector) opera continuamente, liberando recursos que não são mais utilizados. Isso evita erros que poderiam ser cometidos com o gerenciamento direto da memória pelo programador.

Também há um mecanismo eficiente para contornar situações inesperadas que podem ocorrer em tempo de execução. Essas condições excepcionais, chamadas exceções, podem ser devidamente tratadas para tornar as aplicações mais robustas e não permitir que o programa aborte, mesmo frente a situações de erro. Em Java, o tratamento de exceções é parte integrante da própria linguagem e, em alguns casos, é obrigatória.

Um programa Java sempre é verificado antes de ser executado. Essa verificação também é realizada nos browses Web que suportam Java e visa impedir que os applets possam provocar quais quer danos ao computador cliente. Ademais, como Java não permite acesso direto à memória, impede seu uso para desenvolvimento de vírus.

O conjunto desses recursos torna os programas escritos em Java muito mais robustos. Os erros tendem a ser menos freqüentes e sua detecção geralmente ocorre nos primeiros estágios do desenvolvimento, reduzindo o custo e aumentando sua confiabilidade.

Além disso, os programas escritos em Java podem ser executados em todos os sistemas operacionais que contenham um JRE sem a necessidade de modificar uma linha sequer de código. Você pode, por exemplo, escrever e compilar seus aplicativos no Windows e executá-los no Linux e no Solaris.

Em Java, não existem aspectos de implementação dependentes de plataforma. Enquanto em C/C++ um tipo inteiro tem o tamanho da palavra da máquina, por exemplo, em Java um inteiro terá sempre 32 bits. Assim, os byte codes de uma aplicação específica poderão ser transportados para qualquer outra plataforma que suporte Java e executados sem a necessidade de serem recompilados. Essa portabilidade e Java lhe garante uma vantagem indiscutível no ambiente heterogêneo da Internet.

Como os programas Java são compilados para códigos intermediários, próximos às instruções de máquina, eles são muito mais rápidos do que seriam se java fosse simplesmente interpretada.

A tecnologia Java também é mais dinâmica que C/C++. Ela foi projetada para se adaptar facilmente a ambientes em constante evolução (como a Internet). A inclusão de novos métodos e atributos a classes existentes pode ser feita livremente e o tipo de objeto pode ser pesquisado em tempo de execução.

Como se não bastasse tudo isso, Java permite o controle de concorrência e de multiprocessamento (realização de mais de uma tarefa ao mesmo tempo). O resultado disso é o aumento da sensibilidade interativa dos programas e seu comportamento em tempo real.

Os programas em Java podem ter mais de uma linha de execução (thread) sendo lida ao mesmo tempo. O momento de início da execução e a prioridade das linhas de execução podem ser configurados. Assim, o usuário pode continuar interagindo com o programa mesmo quando ele está realizando uma operação demorada em uma linha de execução paralela.

Também são oferecidos recursos de sincronização para as linhas de execução.. A leitura de blocos de instruções críticas pode ser sincronizada para evitar que o movimento assíncrono de threads provoque situações de erro ou quebra de integridade.

Além de todas as vantagens anteriores, Java ainda oferece facilidades para programação de sistemas distribuídos. Sua API contém uma biblioteca de classes e interfaces muito rica para se trabalhar com sockets, TCP/IP, WebServices, etc.

O Ambiente

O ambiente de desenvolvimento Java é composto ou ferramentas e utilitários necessários para realizar as diversas tarefas relacionadas ao desenvolvimento de novos programas, tais como: depurar, compilar e documentar.

O ambiente de execução de Java, por sua vez, é composto somente pelo conjunto de softwares necessários para que seja possível rodar programas já existentes. Isso inclui uma JVM (Java Virtual Machine – Máquina Virtual Java) e bibliotecas de classes e interfaces. A sigla JRE (Java Runtime Environment) é utilizada para representar o Ambiente de execução de Java.

Se você instalar o Java SDK (Software Development Kit – Kit de Desenvolvimento de Software - também chamado JDK), poderá contar tanto com as ferramentas que compõem o ambiente de desenvolvimento de Java quanto com uma versão completa do seu ambiente de execução. Assim, você poderá depurar, compilar, documentar e também executar os programas.

Entretanto, se instalar somente o JRE, não terá as ferramentas necessárias para desenvolver novos programas. Terá apenas uma JVM e as bibliotecas de classes, que possibilitarão a execução de programas já existentes.

Portanto, se o que deseja fazer é escrever, compilar e rodar programas Java, você precisará tanto de um ambiente de desenvolvimento quanto de um ambiente de execução. Nesse case, deve instalar o JDK.

Por outro lado, quando for instalar um programa no computador de outra pessoa (um cliente, por exemplo), possivelmente você desejará somente poder executá-lo. Sendo assim, não serão necessárias as ferramentas de depuração, compilação e documentação. Você somente precisará de uma JVM própria para aquele sistema operacional e da biblioteca de classes de Java. Nesse caso, não será necessário instalar o kit completo de desenvolvimento (o JDK), mas tão-somente o JRE.

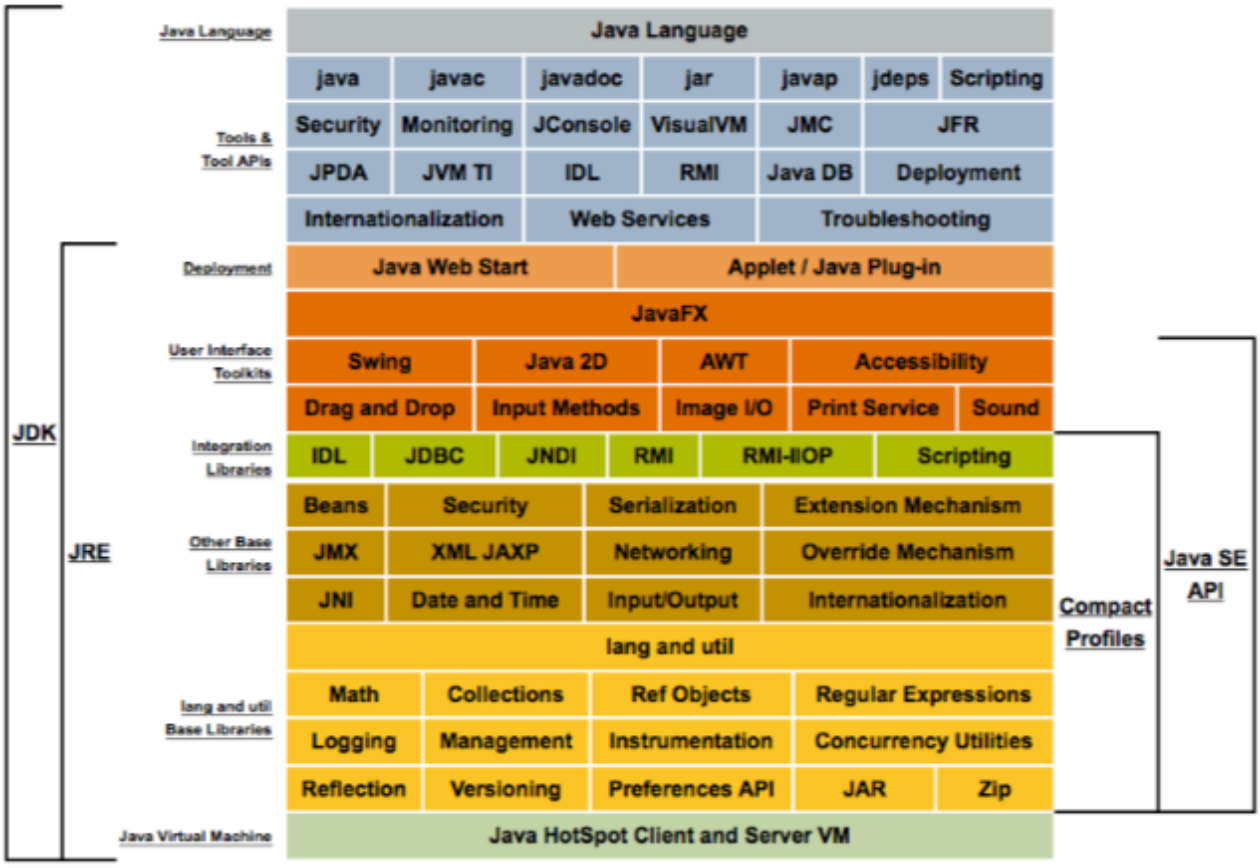
Alguns sistemas operacionais já contêm um ambiente de execução para aplicativos Java, como é o caso do Windows, do Linux e do Solaris. Assim, antes de instalar ou recomendar a instalação do JRE na máquina de outra pessoa para rodar seus aplicativos, verifique se isso é realmente necessário.

Toolkits

A tecnologia Java não se resume a um único toolkit (kit de ferramentas). Ela compõe-se de diversos pacotes com finalidades singulares. A primeira dificuldade enfrentada por quem está iniciando a aprendizagem de Java é exatamente escolher que toolkit instalar. Essa escolha deve levar em conta o que você pretende realizar e que recursos estão disponíveis em cada um deles.

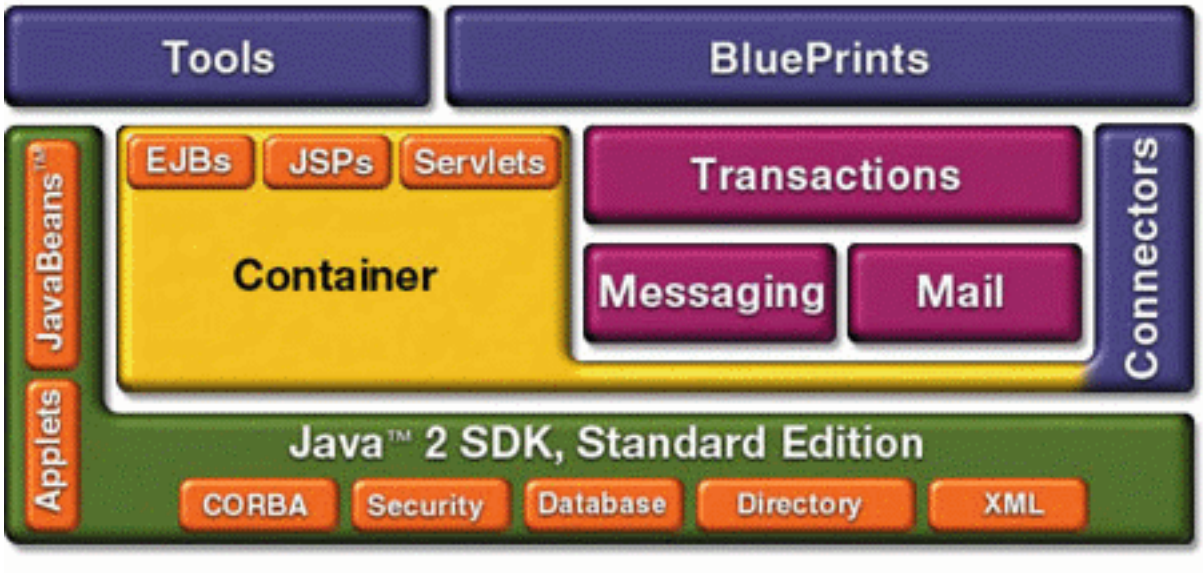
O primeiro passo é compreender a utilidade de cada uma das duas diferentes edições da plataforma Java, quais sejam: Java SE, Jakarta EE. Comece observando, a seguir, o significado de cada uma dessas siglas:

- Java SE: Java Platform Standard Edition – Edição Padrão da Plataforma Java
- Jakarta EE: Jakarta Enterprise Platform – Edição Empresarial da Plataforma Java



A Tecnologia Java SE é a solução adequada para se desenvolver uma grande gama de aplicativos e applets para empresas. Inclui acesso a banco de dados, controle de múltiplas linhas de execução, suporte ao desenvolvimento de aplicações distribuídas e bibliotecas completas para criação de interfaces gráficas, operações em rede e outras.

A tecnologia Jakarta EE e seu modelo de componentes simplificam o desenvolvimento de aplicações para empresas. A plataforma Jakarta EE suporta Web Services necessários para habilitar o desenvolvimento de aplicações de negócio seguras e robustas. É a tecnologia mais adequada para o desenvolvimento de aplicações complexas para rodar na Internet.



Mas a tecnologia Java não se limita às duas edições da plataforma Java. Existem pacotes voltados para finalidades específicas. O Java Card Platform (Plataforma Java Card), por exemplo, é um pacote que se destina a permitir que a tecnologia Java possa rodar em cartões inteligentes e em outros equipamentos eletrônicos com memória limitada.

A tecnologia Java ainda conta com pacotes complementares de suporte ao desenvolvimento para redes de computadores sem fio (Wireless) e também para o uso de XML na criação de aplicações para a Internet ou para equipamentos de consumo.

É claro que o grande número de pacotes que compõem a tecnologia Java apenas demostra a vastidão de oportunidades que estão abertas para aqueles que a dominam. Entretanto, isso tende a confundir quem ainda não está familiarizado com ela.

Configurações

Existem algumas configurações importantes que o instalador do JDK não faz e, por isso, é preciso realizá-las manualmente. Dentre elas, destaca-se a definição de três variáveis de ambiente.

O funcionamento das variáveis de ambiente se assemelha ao das variáveis utilizadas para criação de programas. Mas enquanto as variáveis de um programa são criadas e destruídas junto com ele e só podem ser usadas internamente, as variáveis de ambiente são mantidas pelo próprio sistema operacional e, por isso, as informações nelas contidas podem ser compartilhadas entre diversos aplicativos.

No Windows, um dos modos de se configurar as variáveis de ambiente é através de instruções executadas no prompt de comento. Para abri-lo, vasta ir até o Menu Iniciar/ Programas ? Acessórios e encontrar o atalho para ele.

O prompt de comando não será usado apenas para realizar essas configurações. É através dele que os programas escritos poderão ser compilados e executados, visto que não será utilizado nenhum ambiente de desenvolvimento integrado que inclua uma interface gráfica.

A primeira variável de ambiente a ser configurada chama-se *JAVA_HOME*. Nessa variável deve-se armazenar o diretório raiz onde está instalado o JDK. Caso esse diretório seja `C:\Arquivos de Programas\Java\jdk1.8.0_191`, por exemplo, então a instrução a ser executada no prompt de comando é a seguinte:

```
SET JAVA_HOME=“C:\Arquivos de Programas\Java\JDK1.8.0_191”
```

Ao executar essa instrução, a variável *JAVA_HOME* passará a apontar para o diretório `C:\Arquivos de Programas\Java\jdk1.8.0_191`. É através dessa variável que muitos aplicativos que utilizam recursos da tecnologia Java irão identificar onde eles se encontram.

Outra variável de ambiente que precisa ser configurada chama-se *PATH*. Nela deve ser armazenado o caminho onde se encontram as ferramentas de desenvolvimento de Java. Mas há um cuidado adicional que você precisa tomar ao configurar essa variável: não eliminar o conteúdo que ela já possui.

O Windows armazena na variável *PATH* os caminhos onde serão encontrados os seus principais arquivos de sistema. Assim, o que você deve fazer é adicionar a essa variável o subdiretório bin do JDK, mantendo o seu conteúdo. Isso pode ser feito através da seguinte instrução:

```
SET PATH=%PATH%;%JAVA_HOME%\BIN
```

O primeiro valor atribuído à variável *PATH* é seu próprio conteúdo (%PATH%). É isso que permitirá que ela retenha as referências que já continha. Tipicamente, essa variável armazena um conjunto de caminhos de diretórios separados por ponto-e-vírgula. O segundo valor adicionado a essa variável, indicado após o ponto-e-vírgula, é exatamente o diretório bin do JDK. A expressão %JAVA_HOME% será substituída pelo conteúdo da variável *JAVA_HOME*, definido no passo anterior. Portanto, essa instrução equivale à que segue:

```
SET PATH=%PATH%;”C:\Arquivos de Programas\Java\JDK1.8.0_191\BIN”
```

Se desejar conferir o que está contido nessa variável, basta entrar com a instrução "*PATH*" no prompt de comando.

A última variável de ambiente a ser configurada chama-se *CLASSPATH*. Ela deve apontar para o diretório atual (representado pelo ponto) e também para o subdiretório \jre\lib do JDK. A instrução a ser executada para configurá-la é a que segue:

```
SET CLASSPATH=.;%JAVA_HOME%\JRE\LIB
```

A configuração da variável *CLASSPATH* é indispensável para que o compilador e o interpretador Java encontrem as bibliotecas de classes e de interfaces de que necessitam.

Apesar da aparente dificuldade sentida por quem não está familiarizado com o uso de variáveis de ambiente, observe que o procedimento para a sua configuração é relativamente simples. Mas a configuração das variáveis de ambiente via linha de comando é um procedimento ineficiente, pois você terá de realizar essa tarefa a cada vez que o sistema for inicializado.

No entanto, há como gravar permanentemente a configuração dessas variáveis de ambiente em seu sistema. No Windows, isso pode ser feito no Painel de Controle / Sistema há uma aba chamada Avançado. É nesse local que você poderá configurar todas as variáveis de ambiente do sistema operacional.

Ferramentas

Tendo instalado e configurado o JDK, ainda lhe resta compreender como utilizar as ferramentas básicas aplicáveis ao desenvolvimento de novos programas. Isso pressupõe o uso de um editor de textos para escrever o código e salvá-lo em um arquivo, de um compilador para gerar os byte codes e de um interpretador para executar o programa.

Para melhor compreender o uso das ferramentas de desenvolvimento nas três fases que caracterizam a geração de um novo programa em Java, você deverá escrever, compilar e executar um aplicativo de exemplo. Vale destacar que todos os arquivos-fonte devem ter a extensão java.

Editor

A criação de um novo programa Java começa sempre com a edição de seu código e sua gravação na forma de um arquivo, que deverá ser salvo com a extensão java. O JDK não traz nenhum editor de textos. Mas isso não deverá ser problema algum, uma vez que você pode utilizar um editor de textos comum disponível em seu sistema operacional.

Utilizando o Bloco de Notas ou o WordPad edite o programa Alo.java. Esses são dois editores de texto que fazem parte da instalação básica desse sistema operacional e, apesar de sua simplicidade, têm tudo o que você precisará para criar seus programas.

É aconselhável criar um diretório para armazenar os exemplos editados.”

Alo.java

```
public class Alo {
	public static void main(String[] args) {
		System.out.println(”Alo Mundo Java!”);
	}
}
```

Compilador

Tendo editado e gravado o arquivo Alo.java, você pode passar para a segunda tarefa: compilar esse arquivo. Para isso, você deverá usar o compilador do JDK. Ele se encontra no subdiretório bin e o nome do arquivo é `javac.exe`.

O compilador do JDK não contém nenhuma interface gráfica. Para utilizá-lo você deverá invocá-lo através de instrução executada sua linha de comando. No Windows, poderá ser usado o prompt de comando.

O primeiro passo é posicionar-se no diretório onde se encontra o arquivo Alo.java. Para compilar o arquivo, basta invocar o compilador `javac` seguido do nome completo do arquivo: "`javac Alo.java`". Isso irá criar um novo arquivo no mesmo diretório com o nome Alo.class. Nele serão armazenados os byte codes Java relativos ao código contido no arquivo `Alo.java`.

Se uma mensagem de erro como "*Comando ou nome de arquivo inválido*" for exigida, isso significa que você não configurou corretamente a variável *PATH* no seu sistema. Assim, precisará invocar o compilador utilizando o seu caminho completo, como segue:

```
"C:\Arquivos de Programas\Java\JDK1.8.0_191\bin\javac" Alo.java
```

A segunda tarefa se resume a isso: compilar o arquivo-fonte (Alo.java) para gerar um arquivo contendo os byte-codes Java (`Alo.class`). Diante de mensagens de erro do compilador, revise o texto digitado no arquivo `Alo.java` e tente compilá-lo novamente. Note que o compilador de Java distingue caracteres maiúsculos e minúsculos.

Interpretador

Depois de editar e compilar o programa de exemplo, resta apenas executá-lo. A execução de aplicativos Java é feita por um interpretador. O interpretador que você deverá utilizar é representado pelo arquivo java.exe, localizado no diretório `\bin` do JDK. Ele deverá ser invocado para interpretar o arquivo gerado pelo compilador, qual seja, o arquivo `Alo.class`.

Lembre-se que o arquivo gerado pelo compilador (`Alo.class`) não contém código binário específico para uma plataforma. Ele contém um código intermediário, próximo da linguagem de máquina, que deve ser interpretado por um JRE. Para rodar esse programa você deve utilizar o interpretador java, conforme segue:

```
java Alo
```

A única coisa que esse programa faz e imprimir uma mensagem na tela: "*Alo Mundo Java!*". Caso enfrente dificuldades para executar o arquivo `Alo.class`, revise as instruções para configuração das variáveis de ambiente no tópico anterior. É importante que as variáveis *JAVA_HOME*, *PATH* e *CLASSPATH* estejam configuradas adequadamente para facilitar seu trabalho no que tange à implementação de quaisquer programas Java.

Estrutura de Aplicativos

Aplicativo Java é um programa que pode ser executado em qualquer computador que tenha um JRE instalado. A interação do usuário com um aplicativo pode se dar através de dois modos distintos: textual ou gráfico. No modo textual, o aplicativo irá interagir com o usuário através de um fluxo de entrada e saída de informações na forma de texto simples. No modo gráfico, o usuário irá interagir com o programa através de componentes que contêm uma representação visual (botões, caixas de edição , caixas de checagem, listas, etc.), que são organizados em contêineres e agrupados em janelas.

Para executar um aplicativo não é utilizado nenhum browser (navegador Web). Isso porque sua natureza o distingue dos outros tipos de programas desenvolvidos com Java: os applets e os servlets. Pensar que a tecnologia Java é utilizada somente para gerar programas que rodam na Internet é um erro. Imaginar que ela se volta somente para a criação de applets e de servlets é ignorar parte essencial de seu poder. Você pode utiliza-la para desenvolver aplicativos comerciais que rodem em um único computador ou uma estrutura cliente/servidor para rodar em uma intranet.

Embora Java se destaque na programação para a Internet em função das vantagens de que desfruta nesse ambiente em relação a outras tecnologias, ela também pode ser utilizada para o desenvolvimento de quaisquer aplicativos que possam ser variados em C/C++, Pascal, Cobol ou outra linguagem de programação tradicional. Isso inclui aplicativos que se comunicam com banco de dados locais ou remotos e que interagem com o usuário através de uma interface gráfica.

Para compreender melhor a natureza de um aplicativo, implemente e execute o exemplo que segue:

BemVindo.java

```
import javax.swing.JOptionPane;

public class BemVindo {
	public static void main(String[] args) {
		String nome = JOptionPane.showInputDialog(“Digite seu nome”);
		JOptionPane.showMessageDialog(null, nome + “, seja Bem Vindo”);
	}
}
```

Esse é um exemplo de aplicativo que interage com o usuário através do modo gráfico, utilizando janelas de diálogo para a entrada e saída de informações. Você deve implementá-lo e executá-lo seguindo os mesmos passos já descritos.

Agora é o momento de dissecar o código desse exemplo, realizando uma análise minuciosa do mesmo. É preciso assimilar a função de cada linha de sua estrutura para compreender como são construídos aplicativos com Java.

Declaração Import

A declaração import de Java corresponde à declaração include do C/C++ ou à clausula uses do Pascal. Sua finalidade é importar bibliotecas de classes e interfaces já existentes que serão necessárias para a implementação do arquivo atual.

Fez-se uso de uma declaração import logo na primeira linha do aplicativo implementado. O que ela fez foi importar uma classe, chamada JOptionPane, responsável pela geração e exibição de caixas de diálogo padronizadas para captação de dados e exibição de mensagens na tela do computador.

A classe JOptionPane está localizada dentro de um pacote chamado swing e este, por sua vez, está localizado em um pacote chamado javax. Na importação de uma classe deve ser indicado o caminho completo de sua localização. Isso foi feito com a indicação da classe JOptionPane precedida do pacote em que ela se encontra (swing) e também do pacote no qual o pacote swing se encontra (javax). Os nomes dos pacotes e da própria classe sempre devem ser separador por um ponto.

Nesse momento, você não precisa se preocupar em conhecer a estrutura de pacotes, classes e interfaces que compõem a API de Java. Basta compreender que, para utilizar qualquer recurso dessas bibliotecas em um aplicativo, será preciso importá-lo utilizando a declaração import.

É importante que você se esforce no sentido de memorizar o caminho dos recursos mais utilizados nos exemplos. Sempre que for implementar um novo aplicativo, procure observar atentamente quais foram os recursos importados, pois isso lhe permitirá formar uma idéia prévia acerca de que tarefas ele poderá realizar.

Declaração de Classe

O desenvolvimento de um aplicativo Java se resume à criação de fragmentos de código que se comunicam entre si, chamados classes e interfaces. Como exemplo introdutório, o aplicativo implementado compõe-se de uma única classe. A declaração dessa classe é feita na linha 3.

Toda classe precisa de uma identificação, que representa o nome pelo qual ela será conhecida. A identificação da classe do aplicativo de exemplo é BemVindo e, como pode ser observado, é o último termo de sua declaração.

Note que a identificação da classe BemVindo coincide com o nome do arquivo onde ela fora gravada (BemVindo.java), Isso não é mera coincidência. Ela é uma classe pública e como tal deve estar implementada em um arquivo cujo nome coincida com sua identificação.

É o qualificador public que define a condição de classe pública a uma nova classe. Observe, no exemplo em questão, que o termo public é o primeiro a figurar na declaração da classe BemVindo. Caso não houvesse esse termo na declaração dessa classe, o arquivo em que ela está contida não precisaria se chamar BemVindo.java.

O qualificador public e a identificação da classe BemVindo são separados pelo termo class, que é utilizado na declaração de novas classes. Caso ela não possua nenhum qualificador, então este será o primeiro e único termo a preceder sua identificação.

Método main()

As classes são compostas por atributos e métodos. Esses últimos são blocos de instruções que contêm uma identificação, realizam determinada tarefa e podem retornar algum tipo de informação.

Todo aplicativo é composto por uma ou mais classes e uma delas deve conter um método especial chamado main(). A única classe que compõe o aplicativo de exemplo possui a declaração do método main() e, portanto, ela representa um aplicativo. Você pode observar essa declaração na linha 4.

Se o aplicativo fosse composto de diversas classes, seria aquela que contivesse o método main() que deveria ser executada. O método main() contém as instruções que são lidas quando a classe é executada pelo interpretador Java. Caso você tente executar uma classe que não contém o método main(), será exibida uma mensagem de erro, como segue:

```
Exception in thread 'main' java.lang.NoSuchMethodError: main
```

Essa mensagem lhe indica que não foi encontrado o método main() na classe que você tentou executar e que, portanto, não é possível realizar a operação.

O primeiro elemento da declaração do método main() é o qualificador public. Ele indica que se trata de um método público e determina seu nível de visibilidade. Define que esse método poderá ser invocado a partir de objetos da classe BemVindo que tenham sido criados por outras classes. O que você precisa entender sobre isso, agora, é que o método main() sempre deverá conter o qualificador public em sua declaração.

O segundo elemento é o qualificador static. Ele determina que o método main() é um método estático. Nessa condição, esse método poderá ser invocado a partir da própria classe, sem a necessidade de instanciação de objetos. Esse qualificador também é obrigatório na declaração do método main().

Todo método deve conter em sua declaração a indicação do tipo de retorno. Isso deverá ser inserido antes de seu nome. Quando um método não retorna informação alguma, deve-se incluir o termo void no lugar do tipo de retorno. É isso que deve ser feito no método main(). O termo void deve estar presente para indicar que esse método não retorna qualquer tipo de dado.

O método main() é identificado por esse nome e deverá ser escrito com todas as letras minúsculas. Lembre-se que, em Java, há distinção entre caracteres maiúsculos e minúsculos. assim, se você escrever a identificação desse método com alguma letra maiúscula, o interpretador de Java não o encontrará quando você invocá-lo para executar a classe. O resultado será a exibição da mensagem de erro supracitada.

A declaração do método main ainda contém um último elemento, que aparece envolto em parênteses. Em java, os parâmetros dos métodos têm seu ripo e nome listados dessa forma. No caso em questão args é o nome do parâmetro do método main() e String[] é seu tipo. O tipo desse parâmetro deve ser, necessariamente, este: um vetor de objetos da classe String. Entretanto, seu nome pode ser modificado de args para qualquer outro identificador válido.

Instruções de Blocos

Uma instrução pode ser entendida como um comando que ordena a realização de uma tarefa qualquer. O método main() do aplicativo de exemplo contém três instruções localizadas entre as linhas 5 e 7. São essas instruções que realizam as ações do aplicativo quando a classe BemVindo é executada.

A instrução da linha 5 exibe uma caixa de diálogo solicitando o nome do usuário e armazena essa informação em um objeto chamado st. Observe que esse objeto é do tipo String, utilizado para representar dados textuais em Java.

A instrução da linha 6 exibe uma caixa de diálogo na tela com uma mensagem de boas-vindas, personalizada com o nome do usuário. Perceba que foi utilizado o operador de concatenação (+) para juntar o conteúdo do objeto st com a mensagem complementar: ": seja bem vindo!".

A instrução da linha 7 irá encerrar o aplicativo. Sempre que um aplicativo gerar algum elemento gráfico, ele não será encerrado automaticamente após a leitura da última instrução do método main(). Ao invés disso, é preciso solicitar explicitamente o seu encerramento através da instrução:

```
System.exit(0);
```

Note que essas três instruções encontram-se delimitadas por um par de chaves, localizado nas linhas 4 e 7. O par de chaves, em Java, equivale aos termos Begin e End do Pascal e é utilizado para indicar o início e o fim de um bloco de código. Nesse caso, ele foi utilizado para delimitar o conjunto de instruções que formam o corpo do método main().

O código que compõe o corpo de uma classe também deve estar envolto por um par de chaves. Observe que, nas linhas 3 e 9, há a formação de um par de chaves para delimitar o início e o fim do código que compõe a classe BemVindo.

Argumentos

Você pode passar algumas informações para o aplicativo na instrução em que invoca o interpretador java para executá-lo. Cada conjunto de caracteres, separados por espaço, representará um argumento e poderá ser utilizado para personalizar sua execução.

Neste exemplo, o aplicativo implementado captava o nome do usuário através de uma caixa de diálogo. Mas essa informação poderia ser passada ao aplicativo na mesma instrução que o executa. O próximo exemplo demonstrará como fazer isso. Abra um novo documento em seu editor de textos, digite o programa Argumentos.java, salve-o e compile-o.

Argumentos.java

```
import javax.swing.JOptionPane;

public class Argumentos {
	public static void main(String[] args) {
		String st = args[0];
		JOptionPane.showMessageDialog(null, st + ": seja bem vindo!");
	}
}
```

Para executar esse aplicativo você deve incluir o seu próprio nome ao final da linha de comando: "java Argumentos <nome>".

O nome informado na linha de comando será armazenado na posição zero do parâmetro args do método main() desse aplicativo.

```
String st = args[0];
```

O conteúdo da posição zero desse parâmetro (args[0]) é repassado ao objeto st, que é o primeiro elemento a aparecer na mensagem de boas-vindas.

Desse modo, o resultado obtido é o mesmo que aquele produzido pelo aplicativo anterior. A diferença é que aqui o nome do usuário é repassado como argumento na própria instrução que o executa e no exemplo anterior era captado através de uma caixa de diálogo exibida para o usuário depois que o aplicativo já estava rodando.

Tipos de Dados

Assim como ocorre com as linguagens de programação tradicionais, em Java existem algumas palavras reservadas para a representação dos tipos de dados básicos que precisam ser manipulados para a construção de programas, também conhecidos como tipos primitivos.

Podem-se dividir os tipos primitivos suportados por Java em função da natureza de seu conteúdo. Têm-se quatro tipos para representação de números inteiros, dois ripos para representação de números de ponto flutuante, um tipo para representação de caracteres e um tipo para representação dos valores booleanos (verdadeiro ou falso).

Como você acabar[a percebendo, as milhares de classes disponíveis na API Java também são tipos de dados. Mas enquanto uma classe pode armazenar diversas informações ao mesmo tempo, em seus atributos, e realizar tarefas através de seus métodos, um tipo primitivo pode armazenar somente uma única informação de cada vez e não contém quaisquer métodos para realizar tarefas.

Você também descobrirá que, em Java, não existe um tipo primitivo para representação de texto, como o tipo String da linguagem Pascal. Entretanto, existe uma classe (chamada String) que serve para esse propósito e pode ser usada de modo semelhante a um tipo primitivo e ainda conta com diversos métodos úteis.

Na verdade, também existem classes para representar cada um dos tipos primitivos. Sempre que for preciso realizar uma operação mais complexa com algum tipo de informação, você poderá armazená-la em um objeto da classe competente ao invés de utilizar um tipo primitivo. Assim, poderá fazer uso dos métodos disponíveis nessas classes para realizar diversas operações com o dado armazenado.

Valores Literais

Números inteiros e de ponto flutuante, valores lógicos, caracteres e textos podem ser utilizados em qualquer parte de um programa Java. Mas para escrever um valor de um desses tipos de forma literal no código é preciso representá-lo da forma correta, como pode ser observado na tabela abaixo:

Tipo de dado	Representação	Exemplo
Números inteiros na base decimal	v	11
Números inteiros na base hexadecimal	0xv	0xB
Números inteiros na base octal	0v	013
Números inteiros longos	vL	11L
Números reais de precisão simples	v.vf	24.2f
Números reais de precisão dupla	v.v	24.2
Valores lógicos	v	true
Caracteres	'v'	'H'
Texto	”v”	”Ana Carolina”

VALORES LITERAIS

Na coluna Representação, a letra v foi disposta onde você deve escrever o valor desejado e os demais caracteres representam a parte que não muda. Por exemplo, a representação 0xv, indicada para se escrever números inteiros na base hexadecimal, significa que um valor desse tipo deve sempre ser transcrito precedido do número zero e da letra “x”. Nesse caso particular, não importa se essa letra é minúscula ou maiúscula.

Você pode escrever números inteiros na base decimal em seus programas sem qualquer prefixo ou sufixo. Já a transcrição de números inteiros na base hexadecimal deve obedecer ao padrão descrito no parágrafo anterior, e a transcrição de inteiros na base octal deve sempre ser precedida de um zero. E quando for preciso transcrever um número inteiro longo, deverá ser utilizado o sufixo L.

A maior parte dos números inteiros que você escreverá estará na base decimal. Operações com inteiros na base octal são pouco comuns, mas a realização de operações na base hexadecimal ocorre com certa freqüência.

Os números de ponto flutuante sempre devem ser escritos com um ponto separando sua parte inteira da parte fracionária. Caso não haja parte fracionária, você deve utilizar um zero após o ponto. Além disso, se desejar escrever um número de ponto flutuante utilizando a precisão simples, deve incluir o sufixo F ao final do mesmo. Mas é importante destacar que o uso de precisão simples somente é indicado quando é preciso economizar memória e não há necessidade de precisão superior a sete dígitos na operação realizada.

A transcrição de valores literais para os últimos três tipos de dados não contém as complexidades presentes entre os números inteiros e fracionários. Valores lógicos são transcritos pelas palavras reservadas true e false, um caractere solitário pode ser escrito entre apóstrofes e textos devem ser escritos entre aspas duplas.

Você precisará transcrever alguns desses tipos de dados com muita freqüência para a implementação de programas em Java, como números inteiros na base decimal, números de ponto flutuante de precisão dupla, valores lógicos e textos. Outros tipos serão utilizados com menos freqüência, mas têm importância suficiente para que lhes seja atribuído o devido valor.

O próximo exemplo coloca em prática o que fora descrito sobre a transcrição de valores literais para fixar seu entendimento. Abra um novo documento em seu editor de textos e digite o programa Literais.java, salve, compile e execute.

Literais.java

```
public class Literais {
	public static void main(String[] args) {
		System.out.println(”Inteiro – decimal: ” + 11);
		System.out.println(”Inteiro – hexadecimal: ” + 0xB);
		System.out.println(”Inteiro – octal: ” + 013);
		System.out.println(”Inteiro – longo: ” + 11L);
		System.out.println(”Real – precisão simples: ” + 24.2f);
		System.out.println(”Real – precisão dupla: ” + 24.2);
		System.out.println(”Tipo lógico: ” + true);
		System.out.println(”Caractere: ” + 'H');
		System.out.println(”Texto: ” + “Ana”);
	}
}
```

Este exemplo demostra como você deverá escrever, no código de seus programas, cada um dos tipos de valores literais que foram tratados anteriormente.

Após a execução, observe que todos os números inteiros que você transcreveu no código do programa foram apresentados como sendo 11, mesmo tendo sido escritos como 0xB e 013 nas bases hexadecimal e octal, respectivamente. Isso porque a letra B na base hexadecimal e o número 13 na base octal correspondem, ambos, ao número 11 da base decimal.

Números Inteiros

Os números inteiros são valores, positivos ou negativos, que não possuem uma parte fracionária. Para representá-los em programas Java, você pode escolher um dos quatro tios primitivos listados na tabela abaixo:

Tipo	Tamanho	Mínimo	Máximo
byte	1 byte	-128	127
short	2 bytes	-32.768	32767
int	4 bytes	-2.147.483.648	2.147.483.647
long	8 bytes	-9.223.372.036.854.775.808	9.223.372.036.845.775.807

TIPOS INTEIROS

A coluna Tamanho indica a quantidade de memória consumida por cada um dos tipos inteiros. Note que a opção mais econômica é o uso do tipo byte, que ocupa apenas 1 byte de memória(o equivalente a 8 bits). No entanto, esse tipo de dado somente pode armazenar valores no intervalo entre -128 e 127. Caso precise utilizar valores negativos inferiores a -128 ou valores positivos superiores a 127, terá de escolher outro tipo para sua representação.

O tipo int é certamente, o mais adequado para a maioria das situações. Com ele você pode representar valores positivos e negativos que variam de zero até pouco mais de dois bilhões. Se precisar representar valores que extrapolem esse limite, pode contar com o tio long, com o qual pode manipular números inteiros, positivos e negativos, que variem de zero a pouco mais de nove quintiliões. A transcrição literal de um número inteiro longo deve ser feita utilizando-se a letra L como sufixo

Os tipos byte e short são utilizados, com maior freqüência, para a construção de programas que realizam tarefas onde é preciso manipular uma grande quantidade de dados ao mesmo tempo e em que a quantidade de memória ocupada se torna um problema relevante.

Números de Ponto Flutuante

Os números de ponto flutuante são valores, positivos ou negativos, que podem conter uma parte fracionária. Você poderá representar esse tipo de dado nos programas Java utilizando um dos dois tipos listados na tabela abaixo:

Tipo	Tamanho	Mínimo	Máximo	Precisão
float	4 bytes	-3,4028E+38	3.4028E+38	6-7 dígitos
double	8 bytes	-1,7976E+308	1,7976E+308	15 dígitos

TIPOS REAIS

Apesar de o tipo float ocupar metade da memória consumida por um tipo double, ele é pouco utilizado. isso porque contém uma limitação que compromete seu uso em um número enorme de situações: ele somente mantém uma precisão decimal de 6 a 7 dígitos.

O tipo double [e a opção padrão para a representação de números de ponto flutuante. Além de ter uma precisão superior ao dobro da precisão do tipo float, ele tem capacidade para armazenar valores em um intervalo muito maior. O valor 1,7976E+308 equivale ao número 17976 seguido de mais 304 zeros. Basta lembrar que um quintilhão contém apenas 12 zeros para se ter uma idéia do quão grande é o valor que se pode armazenar no tio double.

Vale ressaltar que os valores mínimo e máximo da Tabela de Tipos reais são aproximados. Eles servem apenas para que se possa compreender a diferença existente entre os dois tios de dados e para orientar sua aplicação.

Lembre-se que, para transcrever um número e armazená-lo como tipo float, você deverá incluir o sufixo F logo após ao mesmo (sem espaço). Por exemplo, se quiser escrever o número 3,14 como float, deverá fazê-lo da seguinte forma: 3.14F. No lugar da vírgula, o ponto é o separador que figura entre a parte inteira e a parte decimal (isso vale também para o tipo double).

Tipos Textuais

É possível representar dois tipos distintos de elementos textuais em Java: caracteres e textos. A representação de um caractere solitário é feita pelo tipo char e a representação de textos é feita pela classe String.

O tipo char representa caracteres de acordo com o padrão Unicode. Esse é um padrão internacional que unificou os caracteres de todos os idiomas escritos do planeta, cujo código ocupa 2 bytes e, por conseguinte, pode representar até 65536 caracteres. Isso é muito mais do que os 256 caracteres permitidos pelo padrão ASCII, cujo código ocupa apenas 1 byte.

A transcrição de um caractere pode ser feita utilizando-se o código hexadecimal do padrão Unicode, que vai de '\u0000' a '\uFFFF'. O prefixo \u indica somente que se trata do código Unicode, e os apóstrofes são sempre utilizados para envolver um caractere que está sendo escrito de forma literal. Você ainda pode se referir a um caractere transcrevendo-o através de seu código decimal.

Existem alguns caracteres especiais que você precisará utilizar quando estiver escrevendo seus programas. A tabela a seguir lista os principais.

Descrição	Unicode	Atalho
Avanço de linha	\u000a	\n
Avanço de parágrafo (tabulação)	\u0009	\t
Retorno de linha	\u000d	\r
Retorno de um espaço (backspace)	\u0008	\b
Apóstrofo	\u0027	\'
Aspas duplas	\u0022	\”
Barra invertida	\u005c	\\

CARACTERES ESPECIAIS

A formatação de mensagens a serem exibidas no vídeo é um exemplo de tarefa que exige, com muita freqüência, alguns dos caracteres listados nessa tabela, como o avanço de linha e de parágrafo.

Enquanto o tipo char representa um caractere, a representação de textos deverá ser feita pela classe String. Essa classe pode ser utilizada de forma similar aos tipos primitivos, mas os valores literais desse tipo são transcritos entre aspas e não entre apóstrofos.

Tipo Lógico

O tipo lógico é representado, em Java, pelo tipo boolean e pode armazenar um de dois valores possíveis: true (verdadeiro) e false (falso). Ele é empregado para realizar testes lógicos em conjunto com operadores relacionais e dentro de estruturas condicionais.

O tipo boolean de Java equivale ao tipo boolean do Pascal. Vale lembrar que em C são existe um tipo lógico. Há somente a convenção de que o valor zero representa falso e que um número diferente de zero representa verdadeiro. Isso também vale para o Visual Basic.

Um tipo lógico foi adicionado ao C++, sob a denominação bool, para armazenar valores true e false. Mas para manter a compatibilidade com o padrão adotado na linguagem C, manteve-se a possibilidade de utilização de números. Em Java, ao contrário, não se pode fazer conversões entre tipos numéricos e valores lógicos.

Variáveis

Uma variável representa a unidade básica de armazenamento temporário de dados e compõe-se de tipo, um identificador (nome) e um escopo. Seu objetivo é armazenar um dado de determinado tipo primitivo para que possa ser recuperado e aplicado em operações posteriores.

Para compreender como funciona o uso de variáveis é preciso analisar como elas são criadas, de que modo recebem e armazenam dados e como estes são recuperados.

Também é importante entender onde elas podem ser declaradas e onde podem ser utilizadas, tendo em vista seu escopo.

A declaração de uma variável instrui o programa a reservar um espaço na memória do computador para que seja possível armazenar um dado de determinado tipo. A quantidade de memória reservada para uma variável é definida com base no seu tipo.

Para uma variável do tipo int, por exemplo, serão reservados 4 bytes (32 bits). Para uma variável do tipo double, por outro lado, serão reservados 8 bytes (64 bits).

A especificação do tipo de dado de uma variável visa determinar quanto de memória deverá ser reservada para ser possível armazenar um valor e impedir que um dado de tipo diferente seja atribuído a ela. Se uma variável do tipo int é declarada, 4 bytes de memória são reservados para ela e não será permitido atribuir-lhe, por exemplo, um valor do tipo double, que precisa de 8 bytes para sua representação.

A importância do identificador da variável também é evidente. É através dele que você irá se referir a ela no código para lhe atribuir algum dado ou para recuperar um dado que fora armazenado. É graças aos identificadores das variáveis que você não precisará se referir diretamente a posições de memória para armazenar e recuperar valores de tipos primitivos.

Imagine uma variável como um apelido para um conjunto de posições da memória do computador onde um dado de determinado tipo é armazenado e pode ser consultado. O seu identificador será utilizado por você para gravar e recuperar dados nas posições de memória para a qual a variável aponta.

Declaração e Inicialização

Em Pascal, a declaração de todas as variáveis de um programa deve ser feita na cláusula Var, que figura antes dos blocos de instruções Begin e End. As variáveis locais de procedimentos e funções, por sua vez, devem ser declaradas após o seu cabeçalho e antes do bloco de instruções. Desse modo, as declarações de variáveis do programa ficam centralizadas em um único lugar do código e as declarações das variáveis de determinado método também ficam agrupadas.

Em Java isso é bem diferente. Uma nova variável pode ser declarada em qualquer parte do corpo de uma classe. Assim, você poderá declarar as variáveis necessárias à realização de uma operação em um local mais próximo de onde fará seu uso. Se bem aplicada, essa facilidade pode proporcionar muito mais legibilidade ao código dos programas.

A declaração de uma variável inicia-se com a indicação do tipo seguido de sua identificação e de ponto-e-vírgula. Veja como poderia ser declarada uma variável para cada um dos tipos de dados estudados anteriormente:

```
byte bt;
short sh;
int it;
long lg;
float fl;
double db;
char ch;
String st;
boolean bl;
```

Foram utilizados nomes curtos para a identificação dessas variáveis. No entanto, isso não é obrigatório. Você pode utilizar nomes extremamente longos e, inclusive, utilizar caracteres acentuados, como segue:

```
byte MinhaVariavelDoTipoByte;
short MinhaVariavelDoTipoShort;
int MinhaVariavelDoTipoInt;
long MinhaVariavelDoTipoLong;
float MinhaVariavelDoTipoFloat;
double MinhaVariavelDoTipoDouble;
char MinhaVariavelDoTipoChar;
String MinhaVariavelDoTipoString;
boolean MinhaVariavelDoTipoBoolean;
```

O nome de uma variável pode ser composto por quaisquer letras e dígitos do padrão Unicode. Lembre-se de que esse padrão prevê elementos de todos os idiomas escritos do planeta e vai muito além dos 256 caracteres do padrão ASCII. Podem ser utilizadas todas as letras do alfabeto de qualquer idioma, números e também símbolos especiais.

Apesar da flexibilidade existente para a escolha de identificadores de variáveis, existem algumas regras que precisam ser observadas. Eles devem iniciar com uma letra, não devem ser usados alguns símbolos espaciais (como + e ©), não podem conter espaços e não podem ser palavras reservadas da linguagem.

O comprimento de identificadores é ilimitado e todos os caracteres que o compõem são significativos. Não obstante, é importante lembrar que a distinção de letras maiúsculas e minúsculas é uma característica inerente à Java e também se aplica aos identificadores de variáveis.

É importante que você observe certos padrões para a definição de identificadores para variáveis. O uso de nomes curtos é indicado para reduzir o volume de código a ser escrito a cada vez que precisar se referir a uma variável e evitar o uso de caracteres acentuadas e símbolos especiais pode ser aconselhável para tornar o programa potencialmente mais potável.

A inicialização de uma variável é feita através da atribuição de um valor literal a ela. Em Java, o símbolo de igualdade (=) é que representa o sinal de atribuição. A atribuição de um valor a uma variável pode ser feita na sua própria declaração, como segue:

```
byte bt = 127;
short sh = 32767;
int it = 2147483647;
long lg = 9223372036854775807L;
float fl = 1.02f;
double db = 5,123456789;
char ch = 'A';
String st = “Meu texto”;
boolean bl = true;
```

Cada uma das linhas anteriores representa uma instrução de atribuição, onde um dado é armazenado em uma variável. Na primeira instrução, por exemplo, o número 127 é armazenado em uma variável do tipo byte, chamada bt. Sa segunda instrução, o número 32767 é armazenado em uma variável do tipo short, chamada sh.

Observe que o valor atribuído à variável do tipo long, chamada lg, contém a letra L como sufixo. Lembre-se que essa é uma convenção sintática que deve ser seguida.

Sempre que for atribuir um dado a uma variável do tipo long, esse valor deve conter o sufixo L. Do mesmo modo, veja que o valor atribuído à variável do tipo float também contém um sufixo (a letra f). O objetivo é o mesmo: indicar o tipo do valor literal.

A declaração e a atribuição de dados a variáveis podem ser feitas em instruções distintas. Você pode escrever uma instrução para declarar uma variável e outra para atribuir-lhe um valor. Veja como se faz isso:

```
int it;
long lg;
it = 2147483647;
lg = 9223372036854775807L;
```

As duas primeiras instruções são declarações de variáveis: a primeira declara uma variável do tipo int, chamada it e a segunda declara uma variável do tipo long, chamada lg. As duas últimas instruções, por sua vez, inicializam essas variáveis. A terceira instrução atribui o valor 2147483647 à variável it e a quarta instrução atribui o valor 9223372036854775807 à variável lg.

É possível, também, declarar variáveis de um mesmo tipo em uma única instrução. Opcionalmente, você também pode inicializar cada uma delas com um valor. Veja como fazer isso:

```
long lg1 = 9223372036854775807L, lg2
double db1 = 10.5, db2 = 15.43;
```

Note que foram declaradas duas variáveis do tipo long em uma única instrução, chamadas lg1 e lg2 e que a primeira delas já foi inicializada na própria declaração. Na segunda instrução, duas variáveis do tipo double, chamada de db1 e db2, são declaradas e ambas são inicializadas.

Você pode, ainda, inicializar uma variável atribuindo-lhe o conteúdo de outra variável já inicializada. Nesse caso, o programa fará uma cópia do dado contido na variável de origem para a posição de memória referenciada pela variável de destino. Observe as instruções a seguir:

```
int it1, it2;
it1 = 55;
it2 = it1;
```

Na primeira instrução são declaradas duas variáveis do tipo int, chamadas it1 e it2. Na segunda instrução, a variável it1 recebe o número 55. Na última instrução, uma cópia do conteúdo da variável it1 é armazenado na variável it2.

Tento compreendido os conceitos básicos sobre a função e o uso de variáveis, resta a implementação de um exemplo prático para fixar esse entendimento. Abra um novo documento em seu editor de texto, digite o programa Inteiros.java, salve, compile e execute o mesmo.

Inteiros.java

```
public class Inteiros {
	public static void main(String[] args) {
		byte bt;
		short sh;
		int it;
		long lg;
		bt = 127;
		sh = 32767;
		it = 2147483647;
		lg = 9223372036854775807L;
		System.out.println(“Limite superior:”);
		System.out.println(“byte:\t” + bt);
		System.out.println(“short:\t” + sh);
		System.out.println(“int:\t” + it);
		System.out.println(“long:\t” + lg);
		System.out.println(“”);
		bt = -128;
		sh = -32768;
		it = -2147483648;
		lg = -9223372036854775808L;
		System.out.println(“Limite inferior:”);
		System.out.println(“byte:\t” + bt);
		System.out.println(“short:\t” + sh);
		System.out.println(“int:\t” + it);
		System.out.println(“long:\t” + lg);
	}
}
```

Esse exemplo procura ilustrar três coisas distintas: como declarar uma variável, como atribuir-lhe um valor e como recuperá-lo. Complementarmente, também demonstra os valores mínimo e máximo que podem ser atribuídos aos quatro tipos de variáveis inteiras.

Para reforçar sua aprendizagem acerca do uso de variáveis, será implementado mais um exemplo. Abra um novo documento em seu editor de textos, digite o programa Reais.java, salve, compile e execute o mesmo.

Reais.java

```
import javax.swing.JOptionPane;

public class Reais {
	public static void main(String[] args) {
		float fl1, fl2;
		double db1 = 5.123456789, db2 = 10.0;
		fl1 = 1.02f;
		fl2 = 2.0f;
		String st = "Valores armazenados:" +
			"\nfl1 = " + fl1 + "\nfl2 = " + fl2 +
			"\ndb1 = " + db1 + "\ndb2 = " + db2;
		JOptionPane.showMessageDialog(null, st);
	}
}
```

Esse exemplo contém alguns elementos que não figuraram no anterior. Um deles é a exibição do resultado em forma de uma caixa de diálogo. A linha de comando é uma ótima opção para a realização de testes, mas o desenvolvimento de aplicativos com interfaces gráficas é muito mais motivador.

Escopo

A idéia de escopo de variáveis está intimamente ligada ao conceito de blocos de instruções, estudados anteriormente. Como fora dito, um bloco é representado por um par de chaves e serve para agrupar diversas instruções. Mas, além disso, ele também define o escopo das variáveis.

O escopo de uma variável é a região do código onde ela é visível, podendo ser referenciada para receber um dado ou para recuperar um valor anteriormente armazenado. O bloco onde uma variável é declarada define seu escopo. Ela somente poderá ser referenciada para atribuir-lhe algum valor ou para recuperá-lo dentro desse bloco. Ela será invisível a todas as outras partes do código, como se não existisse.

Dentro de um mesmo escopo, não pode haver duas variáveis com o mesmo nome. Se a única divisão fosse a própria classe ou o próprio aplicativo, então não seria possível declarar duas variáveis com identificações semelhantes. Como o escopo é dividido em blocos, é possível declarar duas ou mais variáveis com a mesma identificação, desde que em escopos diferentes.

O motivo pelo qual uma variável não pode ser referenciada fora do escopo onde é declarada tem relação com seu ciclo de vida. Ela passa a existir somente quando lhe é reservado um endereço de memória onde possa armazenar seu conteúdo e isso só ocorre quando sua declaração for lida. Além disso, a leitura das instruções do bloco onde a variável se encontra continuará e chegará ao final dele e, então, o endereço de memória que havia sido reservado para essa variável é liberado e ela é destruída. Por isso, qualquer tentativa de se referir a essa variável fora de seu escopo equivale a tentar fazer

referência a algo que não existe, por ainda não ter sido criado ou por já ter sido destruído.

O próximo exemplo traz essas questões conceituais para o âmbito prático e reforça sua compreensão. Abra um novo documento em seu editor de textos, digite o programa Texto.java, salve/compile/execute.

Texto.java

```
import javax.swing.JOptionPane;

public class Reais {
	public static void main(String[] args) {
		String st1, st2 = “Conteúdo da variável ch: “;
		{ char sh = 'A';
			st1 = st2 + ch + “\n”;
		}
		{ char sh = 66;
			st1 = st1 + st2 + ch + “\n”;
		}
		{ char sh = '\u0043';
			st1 = st1 + st2 + ch + “\n”;
		}
		JOptionPane.showMessageDialog(null, st1);
	}
}
```

Apesar de extremamente simples, esse exemplo ilustra como o escopo de uma variável a isola das demais partes do código, permitindo que sejam declaradas diversas variáveis com um mesmo identificador dentro de um escopo comum.

Observe que o escopo maior do aplicativo é definido pelo par de chaves que delimitam o início e o término da definição da classe Texto. O método main(), que se inclui nesse contexto, representa um escopo menor do aplicativo.

Foram declarados três escopos menores dentro do método main. Eles estão isolados entre si e instruções de um desses blocos não podem fazer referência a variáveis de outro. Graças a isso, é permitido que se declarem variáveis com o mesmo nome nesses diferentes contextos.

Entretanto, observe que os objetos st1 e st2 foram referenciados nesses três escopos menores. Isso é possível porque elas foram declaradas dentro de um escopo maior, que é o próprio método main(). Esses dois objetos são utilizados para montar a mensagem que é exibida, na forma de mensagem, ao final do programa.

Conversões

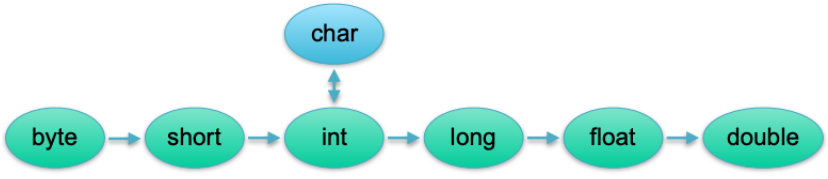
Como já fora dito, é possível atribuir o conteúdo de uma variável a outra. No entanto, se a variável de origem for de um tipo diferente da variável de destino, o dado pode ter de passar por um processo de conversão antes de ser armazenado.

Primeiramente, é preciso compreender duas situações distintas que podem ocorrer no que tange à atribuição do conteúdo de uma variável de tipo numérico para outra variável numérica. Em uma dessas situações não é preciso converter o dado da variável de origem para o tipo da variável de destino. Isso ocorre quando o tipo da variável de destino comporta valores iguais ou superiores aos comportados pela variável de origem. Veja os casos desse tipo listados a seguir:

```
byte → short → int → long → float → double

char → int
```

Esse esquema indica que, para atribuir o conteúdo de uma variável do tipo int a uma variável do tipo long, float ou double, não haverá a necessidade de realizar conversão alguma. Do mesmo modo, uma variável do tipo byte pode ser atribuída a qualquer outro tipo de variável numérica (short, int, long, float e double) sem a necessidade de conversão de seu conteúdo. Isso pode ser feito normalmente, como segue:



No entanto, a atribuição do conteúdo de uma variável de determinado tipo para outro tipo que se encontra à sua esquerda, no esquema anterior, somente será possível mediante conversão explícita desse conteúdo para o tipo de destino. Para atribuir o conteúdo de uma variável do tipo int a uma variável do tipo byte, por exemplo, será preciso convertê-lo para byte. Veja o procedimento necessário:

```
int it = 55;
byte bt = (byte)it;
```

Perceba que o único elemento adicional que é necessário para converter o conteúdo de uma variável para um tipo distinto é incluir esse tipo entre parênteses antes dela. Vale que a segunda instrução converteu o conteúdo de uma variável do tipo int para o tipo byte, possibilitando que ele fosse armazenado na variável chamada bt.

Mas há um problema com esse procedimento que precisa ser considerado. Ao forçar a conversão do conteúdo de uma variável para um tipo que comporta um intervalo de valores inferior ao tipo original, há sempre a possibilidade de esse dado ser corrompido por um processo que pode ser chamado de ajuste circular.

Suponha que haja uma variável do tipo int cujo conteúdo é o número 130 e você resolve convertê-la para o tipo byte com o intuito de armazenar seu valor em uma variável desse tipo, sendo que o maior número comportado por uma variável do tipo byte é 127. É nessa situação que ocorrerá, silenciosamente, o processo de ajuste circular e, ao invés de esta variável receber o número 130, receberá algo bem diferente: o número -126. Veja as instruções que provocariam esse equívoco:

```
int it = 130;
byte bt = (byte)it;
```

A primeira instrução atribui o valor 130 à variável ir. Até este momento não há problema algum, uma vez que o tipo int suporta um valor superior a 2 bilhões. Mas a segunda instrução força a conversão da variável it para o tipo byte, que só suporta valores entre -128 e 127. O número 130 será, então, ajustado para que possa ser armazenado em um tipo byte do seguinte modo:

- Nesse caso, o valor excedente que não pode ser comportado pelo tipo byte é igual a 3 (130 – 127 = 3).
- O número máximo suportado por um tipo numérico incrementado em 1 é equivalente a seu número mínimo, sofrendo o que foi chamado de ajuste circular (127 + 1 = -128). Do mesmo modo, o número mínimo suportado por um tipo numérico decrementado em 1 equivale ao seu limite máximo (-128 – 1 = 127).

- O valor excedente (3) é somado ao limite máximo do tipo byte (127), provocando o ajuste circular (127 + 1 = -128; -128 + 1 = -127; - 127 + 1 = -126).

O que ocorrerá, portanto, é que o valor atribuído à variável bt não será o número 130, contido na variável it, e sim o número -126. Esse é um erro perigoso por que não impede a compilação e a execução do programa. Ele somente será detectado quando você perceber a incoerência que será provocada no resultado da operação.

O próximo exemplo ajudará a melhor compreender as conversões entre tipos numéricos.

ConvEntreNumeros.java

```
import javax.swing.JOptionPane;

public class ConvEntreNumeros {
	public static void main(String[] args) {
		int it1 = 15635;
		long lg = it1;
		float fl = it1;
		short sh = (short)it1;
		double db = 24.75;
		int it2 = (int)db;
		int it3 = (int)Math.round(db);
		String st = "Valores armazenados:" +
			"\nit1 = " + it1 + "\nlg = " + lg +
			"\nfl = " + fl + "\nsh = " + sh +
			"\ndb = " + db + "\nit2 = " + it2 +
			"\nit3 = " + it3;
		JOptionPane.showMessageDialog(null, st);
	}
}
```

Quando você não quiser que os dígitos fracionários de um valor do tipo double sejam ignorados e sim arredondados, pode utilizar o procedimento onde foi atribuído o conteúdo da variável db à variável it3. O método Math.round() arredondou o valor da variável dv, que passou a ser 25, e o retornou na forma de um ripo long. Como não pode ser atribuído um tipo long a um tipo int, mantém-se o conversor (int) para que esse valor possa ser atribuído à variável it3.

Agora que já foram analisadas as conversões entre tipos numéricos, é preciso compreender outro tipo de conversão que precisa ser realizada com freqüência: a conversão de textos em números e vice-versa. Para realizar essas operações será preciso utilizar alguns métodos das classes Integer, Float, Double e String.

Suponha que você tenha o seguinte texto:

```
String st = “15”;
```

Se você quiser atribuir esse texto a uma variável de tipo numérico, deverá converte-lo para o tipo da variável de destino. Veja como fazer isso:

```
byte bt = (byte)Integer.parseInt(st);
short sh = (short)Integer.parseInt(st);
int it = Integer.parseInt(st);
long lg = Integer.parseInt(st);
float fl = Float.parseInt(st);
double db = Double.parseInt(st);
char ch = (char)Integer.parseInt(st);
```

Cada uma das instruções anteriores ilustra a forma correta para se atribuir o texto a uma variável de tipo numérico, realizando a sua conversão para o tipo respectivo. Para atribuí-lo a uma variável do tipo int, por exemplo, utiliza-se o método parseInt() da classe Integer, como segue: Integer.parseInt(st). Essa instrução irá converter o texto em um tipo int para que possa ser armazenado na variável it.

A instrução Integer.parseInt(st) será utilizada para atribuir o texto a quaisquer variáveis de um dos tipos inteiros (byte, short, int e long). Mas como ela produz um valor do tipo int, se for atribuí-lo a uma variável do tipo byte, short ou char, deve-se convertê-lo para o tipo de destino, como fora ilustrado no exemplo acima.

Para converter o texto em um tipo float e armazená-lo na variável fl, basta utilizar o método parseFloat() da classe Float do modo que fora apresentado. O mesmo acontece com a conversão para o tipo double.

O próximo exemplo procura ilustrar como se faz a atribuição de textos a variáveis de tipo numérico.

ConvTextoNumeros.java

```
import javax.swing.JOptionPane;

public class ConvTextoNumeros {
	public static void main(String[] args) {
		String st;
		st = JOPtionPane.showInputDialog("Digite um número");
		double db = Double.parseDouble(st);
		float fl = Float.parseFloat(st);
		long lg = Integer.parseInt(st);
		int it = Integer.parseInt(st);
		short sh = (short)Integer.parseInt(st);
		byte bt = (byte)Integer.parseInt(st);
		char ch = (char)Integer.parseInt(st);
		String st = "Valores armazenados:" +
			"\ndb = " + db + "\nfl = " + fl +
			"\nlg = " + lg + "\nit = " + it +
			"\nsh = " + sh + "\nbt = " + bt +
			"\nch = " + ch;
		JOptionPane.showMessageDialog(null, st);
	}
}
```

Se a transferência de um texto para uma variável de tipo numérico é relativamente complexa, o caminho inverso mais simples. Para atribuir o conteúdo de uma variável de tipo numérico a um objeto da classe String, será utilizado sempre o mesmo método, como segue:

```
st = String.valueOf(bt);
st = String.valueOf(sh);
st = String.valueOf(it);
st = String.valueOf(lg);
st = String.valueOf(fl);
st = String.valueOf(db);
st = String.valueOf(ch);
```

O método que converte o conteúdo de uma variável de qualquer tipo numérico em um texto é o método valueOf() da própria classe String. Assim, tomando-se uma variável numérica qualquer, é possível converter seu conteúdo para uma forma textual através da instrução: String.valueOf().

O próximo exemplo ilustra como converter o conteúdo de variáveis dos dois tipos numéricos mais usados (int e double) para texto.

ConvNumerosTexto.java

```
import javax.swing.JOptionPane;

public class ConvNumerosTexto {
	public static void main(String[] args) {
		double db = 1536.67;
		int it = 650;
		String st1 = String.valueOf(db);
		String st2 = String.valueOf(it);
		String st = “Valores armazenados:” +
			“\nst1 = “ + st1 + “\nst2 = “ + st2;
		JOptionPane.showMessageDialog(null, st);
	}
}
```

Observe que o procedimento utilizado para a conversão dos valores contidos nas variáveis dos tipos int e double para texto é o mesmo. Ele também é o mesmo para se converter valores de variáveis de outros tipos numéricos para texto.

Constantes

O uso de constantes é menos freqüente que o uso de variáveis. No entanto, há situações em que elas são requeridas e, por isso, é indispensável entender o que são, para que servem e como podem ser utilizadas.

A declaração de uma constante contém apenas um elemento a mais que a declaração de uma variável: a palavra reservada final. Assim como as variáveis, as constantes compõem-se de um tipo, um identificador e um escopo.

Mas enquanto a declaração e a inicialização de variáveis podem ser feitas em instruções distintas, toda constante deve ser declarada e inicializada em uma única instrução e, depois disso, não lhe pode ser atribuído outro valor. É em função disso que o valor recebido por uma constante nunca muda.

Existe uma palavra reservada em Java chamada const, mas não tem uso definido. Por isso, não se confunda: para declarar uma constante deverá ser utilizada sempre a palavra reservada final.

O uso mais comum de constantes é sua declaração como atributos de classe, Desse modo, o valor que cada constante armazena ficará disponível para ser recuperado por todos os métodos da classe. Caso uma constante seja declarada dentro do bloco de instruções de um modo, seu escopo será limitado a esse bloco e não poderá ser utilizada pelos demais métodos. Para que fique disponível a todos os métodos da classe, ela deve ser declarada no bloco que delimita o corpo da própria classe.

Pela sua própria natureza, as constantes armazenam valores que jamais mudam. Assim, não faz sentido declarar uma constante em diferentes classes que precisem utilizá-la. Faz muito mais sentido declará-la em uma única classe e torná-la acessível às demais classes. Isso é feito utilizando o qualificador static.

Além de tudo o que fora dito acerca das constantes, é importante observar uma convenção no momento de declará-las: devem-se utilizar somente letras maiúsculas para seu identificador. A observância dessa convenção torna o código-fonte muito mais legível, facilitando a distinção entre variáveis e constantes.

Você pode declarar uma constante dentro de um método qualquer, mas, como fora dito, isso não faz sentido. O procedimento correto é a criação de uma ou mais classes para armazenar valores constantes e torná-los acessíveis às outras classes. O próximo exemplo ilustra como isso é feito.

Constantes.java

```
public class Constantes {
	static final double COFINS = 0.03;
	static final double PIS = 0.0065;
}
```

As duas constantes declaradas e inicializadas são do tipo double e contêm o qualificador static. Lembre-se que é esse qualificador que tornará essas constantes acessíveis às demais classes a partir da própria classe Constantes, como será demonstrado.

Agora é preciso implementar outra classe para compreender como o valor armazenado por essas constantes pode ser utilizado em operações realizadas em outras classes. O código a seguir contém a implementação da classe Receita, que ilustra como as constantes COFINS e PIS, declaradas na classe Constantes, podem ser utilizadas.

Receita.java

```
public class Receita {
	public static void main(String[] args) {
		double bruta = 15000.0;
```

```
            double cofins = bruta * Constantes.COFINS;
            double pis = bruta * Constantes.PIS;
            double liquida = bruta – cofins – pis;
            String st = “Receita bruta: \t\t” + bruta +
                  “\n(-) COFINS: \t\t” + cofins +
                  “\n(-) PIS: \t\t” + pis +
                  “\n(=) Receita liquida: \t” + liquida;
            System.out.println(st);
      }
}
```

É importante perceber como as constantes PIS e COFINS, declaradas na classe Constantes, foram utilizadas para realizar uma operação de cálculo na classe Receita. Do mesmo modo como foram utilizadas nessa classe, poderiam ser utilizadas em quaisquer outras classes.

Elementos Léxicos

Existem alguns elementos na linguagem Java com os quais você irá se deparar com muita freqüência. São aquelas palavras e símbolos que possuem alguma função específica.

Neste tópico, alguns desses elementos são tratados de forma sucinta. O objetivo não é oferecer uma explicação detalhada de qualquer um deles, mas somente apresenta-los a você. O conhecimento mais profundo de seu uso se desenvolverá, naturalmente, à medida que eles forem sendo utilizados para a implementação de exemplos.

Não são abordados todas as palavras e símbolos que possuem função específica na linguagem, mas somente elementos básicos e essenciais. Para efeito didático, eles foram agrupados em palavras reservadas, identificadores, separadores e comentários.

Palavras Reservadas

As palavras reservadas são identificadores que possuem uma função específica e são essenciais para a escrita dos programas. A junção das palavras reservadas com os operadores e separadores formam a base da linguagem Java.

Toda e qualquer palavra reservada somente pode ser utilizada para o seu propósito. Não pode, por exemplo, ser utilizada como identificador de variável, constante, método ou classe.

A tabela a seguir contém as 50 principais palavras reservadas da linguagem Java. Não se preocupe em entender de imediato a função de cada uma delas. Você aprenderá. Gradualmente, como utilizá-las.

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

PRINCIPAIS PALAVRAS RESERVADAS

Em sua versão 1.0 a linguagem Java possuía 59 palavras reservadas. Entretanto, algumas delas não eram usadas, não foram divulgadas e nem utilizadas nas implementações posteriores. São exemplos disso: byvalue, cast, fufute, generic, inner, operator, outer, rest e var. Este último reaparecendo no Java 10

Identificadores

Os identificadores são utilizados como nome para classes, objetos, atributos, métodos, variáveis e constantes. Eles podem ser compostos por qualquer seqüência de letras minúsculas e maiúsculas, números e caracteres de sublinhado. Entretanto, não podem começar com um número, para que não sejam confundidos com os literais numéricos.

É sempre importante ter em mente que Java distingue as letras minúsculas das maiúsculas. Um identificador chamado Total será distinto de outro chamado TOTAL. Por isso, a adoção de um padrão para o uso de maiúsculas e minúsculas é indispensável.

Há uma convenção utilizada pela Sun para nomear os identificadores. Para atributos públicos, métodos públicos e parâmetros de métodos são utilizados identificadores que iniciam com letra minúscula e toda palavra subseqüente que o compõe tem sua primeira letra maiúscula. Por exemplo: receitaLiquida, valorEmCaixa e descontoConcedido.

Para identificadores de atributos privados, métodos privados e variáveis locais utilizam-se apenas letras minúsculas e as palavras são separadas por sublinhados. Por exemplo receita_liquida, valor_em_caixa e desconto_concedido.

Os identificadores de constantes, por outro lado, são escritos somente com letras maiúsculas como forma de destacá-las entre as variáveis. Por exemplo: COFINS, VLR_PI, RGB_AMARELO E CM_POR_POLEGADA.

Você não precisa adotar a convenção supracitada para definir os identificadores de seus programas. Ela apenas serve como um exemplo ilustrativo que coloca em relevo a importância de se estabelecer, conscientemente, um padrão para o uso de letras maiúsculas e minúsculas.

Separadores

Como pode ser pressuposto, com base no próprio nome, os separadores são caracteres especiais utilizados na linguagem Java para separar e, ao mesmo tempo, manter o vínculo entre dois elementos.

A tabela a seguir contém o símbolo e o nome dos seis separadores mais utilizados. Seu uso é freqüente na construção de quaisquer programas e, por isso, é importante realizar uma análise pormenorizada dos mesmos.

Símbolo	Nome
.	Ponto
,	Virgula
;	Ponto-e-vírgula
()	Parênteses
{}	Chaves
[]	Colchetes

RELAÇÃO DE SEPARADORES

O ponto é utilizado para separar identificadores de classes e objetos de seus métodos e atributos. Ele também é aplicado para separar pacotes, sub-pacotes e classes em declarações import.

A vírgula serve para separar os identificadores em declaração de múltiplas variáveis em instrução única. Ela ainda separa os parâmetros na definição e invocação de métodos.

O ponto-e-vírgula indica o final de uma instrução. Ele é utilizado constantemente, uma vez que as instruções são o elemento básico e fundamental de quaisquer programas. Além disso, ela também separa as três declarações do cabeçalho do laço for.

Os parênteses definem abordem de precedência em quaisquer tipos de expressões. Caso queira que uma soma seja efetuada antes de uma operação de multiplicação, por exemplo, basta delimitá-la com parênteses. Eles também servem para delimitar parâmetros na definição e invocação de métodos.

As chaves delimitam o bloco de código que compõe cada classe, método e escopo local. Em estruturas de seleção e de repetição que contém mais de uma instrução, as chaves são utilizadas para agrupar o conjunto de valores a ser atribuído a um vetor ou a uma matriz.

Os colchetes, por seu lado, são usados na declaração de vetores e matrizes. Eles também são usados para definir a quantidade de posições desses elementos e para fazer referência a uma posição especifica dos mesmos.

Comentários

Os comentários são observações e notas sobre o programa e são escritos diretamente no seu código-fonte. Eles são ignorados pelo compilador e não produzem qusquer efeitos sobre a execução. No entanto, do ponto de vista da engenharia de software, eles têm função fundamental.

Conforme a complexidade dos programas aumenta e o tempo passa, até mesmo o próprio desenvolvedor passa a sentir dificuldades para compreender seus algoritmos. Algumas notas bem colocadas sobre a função e funcionamento de seus elementos essenciais podem ser vitais para evitar esse tipo de transtorno.

Quando se trabalha em equipe, a importância dos comentários aumenta. Nesse caso, é vital que se estabeleçam padrões para seu uso, do modo que todos os membros possam compreender rapidamente os comentários escritos pelos demais.

Existem três tipos distintos de comentários:

- De uma única linha.
- De múltiplas linhas.
- De documentação.

Os comentários de uma única linha começam com duas varras (//) e se estendem até o final da linha. São geralmente utilizados para a inclusão de notas breves que não ultrapassem uma linha. Tudo o que estiver depois das duas barras é ignorado pelo compilador. Veja seu uso para o comentário de uma única instrução:

```
System.out.println(st); // Imprime o texto contido em st
```

Os comentários de múltiplas linhas iniciam-se com uma barra seguida de um asterisco (/*) e terminam com um asterisco seguido de uma (*/). São utilizados para delimitadas comentários mais longos, que ocupam duas ou mais linhas. O conteúdo que estiver entre os sinais /* e */ será ignorado pelo compilador. Veja um exemplo:

```
/*
      Esse método realiza três operações distintas:
      1) Valida os valores contidos nos parâmetros de entrada.
      2) Calcula a média aritmética simples desses valores.
      3) Arredonda e retorna o resultado.
*/
public static int media(String st1, String st2) {
      try {
            double db1 = Double.parseDouble(st1);
            double db2 = Double.parseDouble(st2);
            double db3 = (db1 + db2) / 2;
            return (int)Math.round(db3);
      } catch(Exception e) { return 0; }
}
```

Os comentários de documentação são uma forma especial de comentário. Eles são utilizados por uma ferramenta que acompanha o J2SDK, chamada javadoc, para gerar a documentação para as classes que compõem o programa. Essa ferramenta varre o código-fonte do programa em busca esses comentários especiais e gera uma série de documentos HTML contendo a descrição de pacotes, classes, construtores, atributos e métodos. Essa documentação será gerada com base nos mesmos padrões adotados na documentação da API do próprio J2SDK.

O símbolo que indica o início de um comentário de documentação é uma barra seguida de dois asteriscos (/** e o encerramento desse comentário é feito com o mesmo símbolo utilizado para comentários longos, ou seja, um asterisco seguido de uma barra (*/).

A ferramenta javadoc reconhece diversas tags, escritas no corpo de um comentário de documentação e precedidas do sinal de arroba (@). Veja a descrição de cada uma dessas tags:

@author: inclui uma nota sobre o autor se a opção -autor for usada para a execução da ferramenta javadoc.

- @param: usada para descrever parâmetros de métodos e construtores.
- @return: usada para descrever o tipo de retorno de métodos.
- @see: adiciona um link para classes e métodos relacionados.

- @throws: especifica exceções disparadas por um método.
- @exception: tem a mesma função que a tag @throws.
- @deprecated: indica que determinado elemento da classe não deve mais ser utilizado.
- @link: permite incluir, manualmente, um link para um documento HTML.
- @since: utilizada para indicar em que versão o recurso foi introduzido.
- @version: adiciona uma nota sobre a versão da classe ou método.

Para ter uma idéia mais concreta sobre comentários de documentação, será implementada a classe Documentacao.

Documentacao.java

```
import javax.swing.JOptionPane;
/**
  * Essa classe visa ilustrar o uso e comentários comuns
  * de documentação.
  * @author Rui Rossi dos Santos
  * @ version 1
*/
public class Documentacao {
      /**
      * Armazena um texto qualquer
      * @see java.lang.String
      */
      private String texto;
      /**
      * Construtor padrão da classe. Ele atribui o conteudo
      * de seu parâmetro Texto ao atributo Texto da classe.
      * @param Texto O texto a ser atribuido ao atributo Texto
      */
      public Documentacao(String texto) {
            this.texto = texto;
      }
      /**
      * Esse método inverte o conteúdo do atributo Texto.
      * @return Um objeto <code>String</code> com o conteúdo
      * invertido do atributo Texto.
      */
```

```
	public String inverso() {
		String st = "";
		for(int i = 0;i < Texto.length();i++)
			st = Texto.charAt(i) + st;
		return st
	}
	/**
	* Método responsável pela inicialização do programa.
	* Cria a instância da classe Documentacao e a usa para
	* exibir o inverso da palavra ROMA em uma caixa de diálogo.
	* @see javax.swing.JOptionPane
	*/
	public static void main(String[] args) {
		Documentacao dc = new Documentacao("ROMA");
		String st = dc.inverso();
		JOptionPane.showMessageDialog(null, st);
	}
}
```

Para gerar a documentação dessa classe, basta posicionar-se no diretório onde você salvou o arquivo Documentacao.java e executar a seguinte instrução no prompt de comando:

```
javadoc Documentacao.java -d c:\docs -autor
```

Essa instrução executa a ferramenta javadoc para extrair os comentários de documentação do arquivo Documentacao.java, gerando os arquivos HTML no diretório c:\docs. Serão criados 12 arquivos HTML.

Se você omitir a opção "-d c:\docs", esses arquivos serão criados no diretório corrente. A opção "-autor" simplesmente indica que devem ser incluídos os comentários relativos à autoria da classe.

Você também pode utilizar a opção "-link". Ela faz com que todas as referências a classes e interfaces da API do JDK, contidas nessa classe, sejam convertidas em links para as páginas HTML onde se encontra a documentação correspondente. Nesse caso, a instrução deveria ser escrita do seguinte modo:

```
javadoc Documentacao.java -d c:\docs -autor -link
                              "C:\Arquivos de Programas\Java\JDK1.8.0_191"
```

Para visualizar a documentação gerada para a classe Documentacao, basta abrir o arquivo Documentacao.html.

Os demais arquivos gerados pela ferramenta javadoc somente têm importância quando você tiver gerado a documentação de duas ou mais classes ou interfaces. Para gerar a documentação de todas as classes do diretório c:\java\Projetos basta entrar com a instrução a seguir:

```
javadoc *.java -d c:\docs -autor -link
                                "C:\Arquivos de Programas\Java\JDK1.8.0_191"
```

Para visualizar um índice contendo a relação de todas as classes para as quais fora gerada a documentação, abra o arquivo index.html. Do lado esquerdo, haverá links para todas as classes documentadas. Clicando sobre um deles, a página contendo a documentação da classe será aberta à direita.

Operadores

Os operadores são caracteres especiais utilizados para realizar determinadas operações com um ou mais operandos. A operação é determinada pelo tipo de operador utilizado e os operandos assumem o papel de argumentos na mesma, podendo ser valores literais, variáveis, constantes ou expressões.

Com base no número de operandos assumidos pelos operadores, é possível classificá-los cm operadores unários, operadores binários e operadores ternários. Os operadores unários realizam operações com um único operando, os operadores binários realizam operações sobre dois operandos e os operadores ternários realizam operações envolvendo três operandos.

Para facilitar o entendimento, os operadores podem ser agrupados com base no tipo de operação que realizam. Desse modo, haverá três tipos básicos de operadores: aritméticos, relacionais e lógicos. Nos tópicos subseqüentes, será explicado o uso dos operadores que compõem cada um desses grupos.

É importante ter em mente que certos operadores podem se comportar de maneira diferente quando são utilizados com diferentes tipos de operandos. É o caso do sinal de adição (+), que serve para realizar operações de soma com operandos numéricos e também pode ser usado para concatenação de textos.

Aritméticos

Os operadores aritméticos são utilizados para a realização de operações matemáticas com operandos de tipos numéricos, do mesmo modo como são aplicados na álgebra. A tabela a seguir relaciona os operadores aritméticos disponíveis.

Operador	Função
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto de divisão)
++	Incremento
--	Decremento
+=	Atribuição aditiva
-=	Atribuição subtratia
*=	Atribuição de multiplicação
/=	Atribuição de divisão
%=	Atribuição de módulo

OPERADORES ARITMÉTICOS

Os operadores aritméticos não podem ser utilizados com operandos de tipo booleano (boolean) ou com texto. No entanto, podem ser aplicados como caracteres (char), uma vez que esse é um subconjunto dos números inteiros.

Segue exemplos da utilização dos operadores aritméticos:

```
int it = 9 % 2; // calcula o resto da divisão
int it1 = 2 * (1 + 3);
int it3 = ++it1; // efetua o incremento de it1 e armazena em it3
int it4 = it3++; // armazena o conteúdo de it3 em it4 e após incremento em it3
val += 2; // soma 2 ao conteúdo de val
```

O próximo exemplo servirá para fixar o entendimento da função e do uso de cada um dos operadores aritméticos.

OperadoresAritmeticos.java

```
public class OperadoresAritmeticos {
	public static void main(String[] args) {
		double db1, db2;
		db1 = 5;
		db2 = 3;
		System.out.println(“Valores iniciais das variáveis:”);
		System.out.println(“db1: ” + db1);
		System.out.println(“db2: ” + db2);
		System.out.println(“”);
```

```
            System.out.println("Operações aritméticas básicas:");
            System.out.println("Soma (db1 + db2): \t\t");
            System.out.println(db1 + db2);
            System.out.println("Subtracao (db1 – db2): \t\t");
            System.out.println(db1 - db2);
            System.out.println("Multiplicação (db1 * db2): \t\t");
            System.out.println(db1 * db2);
            System.out.println("Divisão (db1 / db2): \t\t");
            System.out.println(db1 / db2);
            System.out.println("Módulo (db1 % db2): \t\t");
            System.out.println(db1 % db2);
            System.out.println("");
            System.out.println("Incremento e decremento:");
            db1++;
            db2--;
            System.out.println("db1++: " + db1);
            int val = 2;
            System.out.println("db2--: " + db2);
            System.out.println("");
            System.out.println("Operações de atribuição:");
            System.out.println("Atribuição aditiva: (db1 += 2): \t\t");
            System.out.println(db1 += 2);
            System.out.println("Atribuição subtrativa: (db1 -= 3): \t");
            System.out.println(db1 -= 3);
            System.out.println("Atribuição de multiplicação: (db1 *= 2):\t");
            System.out.println(db1 *= 2);
            System.out.println("Atribuição de divisão: (db1 /= 2): \t");
            System.out.println(db1 /= 2);
            System.out.println("Atribuição de módulo: (db1 %= 2): \t");
            System.out.println(db1 %= 2);
      }
}
```

Relacionais

Os operadores relacionais são utilizados para comparar a igualdade e a ordem entre valores literais, variáveis, constantes e expressões. Todas as operações realizadas com esse tipo de operadores envolvem dois operandos e retornam um valor lógico true (verdadeiro) ou false (falso). A tabela a seguir resume os operadores relacionais disponíveis.

Operador	Função
==	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a

OPERADORES RELACIONAIS

Os operadores de igualdade (==) e de desigualdade (!=) podem ser aplicados para comparar valores de quaisquer tipos primitivos (byte, short, int, long, float, double, char e boolean). Observe as instruções a seguir:

```
int it1 = 10, it2 = l2, it3 = 10;
boolean bl1 = it1 == it2;
boolean bl2 = it1 == it3;
boolean bl3 = it1 != it2;
```

A primeira instrução declara e inicializa três variáveis do tipi inr, chamadas it1, it2 e it3. às variáveis it1 e it3 é atribuído o número 10 e à variável it2 é atribuído o número 12. A segunda instrução atribui à variável booleana, chamada bl1, o resultado da comparação de igualdade entre it1 e it2. Como it1 e it2 contêm valores diferentes, será atribuído o valor false à variável bl1. A terceira instrução realiza uma comparação de igualdade entre as variáveis it1 e it3, armazenando o resultado dessa expressão na variável booleana chamada bl2. Como it1 e it3 contém o mesmo valor (dez), a variável bl2 receberá o valor true. A terceira instrução faz um teste de desigualdade entre as variáveis it1 e it2, armazenando o resultado na variável booleana chamada bl3. Como it1 e it2 contém valores diferentes, será atribuído o valor true à variável bl3.

Não é aconselhável utilizar os operadores de igualdade e de desigualdade para comparar textos representados por objetos da classe String. Estranhamente, se você comparar duas Strings contendo o mesmo texto com o operador de igualdade, poderá acabar obtendo um resultado false, que indica que eles não são iguais. Observe as instruções a seguir:

```
String st1 = new String(“abc”);
String st2 = new String(“abc”);
boolean bl = st1 == st2;
```

Nas duas primeiras linhas são declarados e inicializados dois objetos da classe String, chamados st1 e st2. A ambos é atribuído o texto "abc". Na terceira instrução, faz- se uma comparação de igualdade entre st1 e st2 e o resultado dessa expressão é armazenado em uma variável booleana, chamada bl. Embora contenham o mesmo texto, o resultado dessa comparação será false. Isso porque, quando aplicados com operandos que são objetos e não tipos primitivos, os operadores de igualdade e de desigualdade verificam se ambos os operandos ocupam o mesmo endereço de memória para decidir se são ou não iguais. No case em questão, os objetos st1 e st2, embora tenham o mesmo texto, ocupam posições distintas na memória e, por isso, serão avaliados como elementos diferentes.

Para verificar se dois textos são iguais, a forma mais adequada é invocar o método equals() de um deles e utilizar o outro como argumento. Veja como isso pode ser feito:

```
String st1 = new String("abc");
String st2 = new String("abc");
boolean bl = st1.equals(st2);
```

Nesse caso, é invocado o método equals() do objeto st1 para avalias se o texto nele contido é igual ao texto armazenado no objeto st2. O retorno dessa expressão será armazenado em uma variável booleana e será true, tendo em vista que fora armazenado o mesmo texto em ambos.

Comparações com operadores relacionais de ordem só podem ser feitas com operandos de tipos numéricos (byte, short, int, long, float, double e char). Pode se comparar operandos inteiros, de ponto flutuante e caracteres para saber qual é o maior ou menor. Entretanto, não é possível realizar esse tipo de comparação com operandos do tipo boolean, tendo em vista que seus valores não possuem um ordenamento. Veja como são realizadas comparações de ordem:

```
byte it1 = 10, it2 = 12, it3 = 10;
boolean bl1 = it1 > it2;
boolean bl2 = it1 < it2;
boolean bl3 = it1 >= it2;
boolean bl4 = it1 <= it2;
```

O exemplo seguinte contém instruções que utilizam todos os operadores relacionais disponíveis e, certamente, o ajudará a entender melhor como deve utiliza-los.

OperadoresRelacionais.java

```
public class OperadoresRelacionais {
	public static void main(String[] args) {
		int it1, it2;
		it1 = 5;
		it2 = 3;
		System.out.println(“Valores das variáveis:”);
		System.out.println(“it1: ” + it1);
		System.out.println(“it2: ” + it2);
		System.out.println(“”);
		System.out.println(“Comparações entre as variáveis:”);
		System.out.println(“it1 == it2:\t”);
		System.out.println(it1 == it2);
		System.out.println(“it1 != it2:\t”);
		System.out.println(it1 != it2);
		System.out.println(“it1 > it2:\t”);
		System.out.println(it1 > it2);
		System.out.println(“it1 < it2:\t”);
		System.out.println(it1 < it2);
		System.out.println(“it1 >= it2:\t”);
		System.out.println(it1 >= it2);
		System.out.println(“it1 <= it2:\t”);
		System.out.println(it1 <= it2);
	}
}
```

Lógicos

Os operadores lógicos são utilizados para construir expressões que retornam um resultado booleano (true e false). Com exceção dos operadores if-then-else ternário e NOT unário lógico, todos os demais envolvem dois operandos, que podem ser valores literais, variáveis, constantes ou expressões que resultem em um valor booleano. Esses operadores são sempre aplicados em conjunto com dois operandos booleanos em uma operação da qual resultará um novo valor booleano.

Pode ser difícil compreender esses operadores fora do contexto em que são aplicados. Quando estiverem sendo usados em estruturas de decisão e de repetição ficará mais clara a função de cada um deles. Neste momento, não se preocupe em desenvolver um entendimento amplo e prático sobre eles. Procure somente formar uma idéia superficial do seu significado e finalidade. Mais tarde, você pode retornar a esse tópico para reforçar o seu entendimento.

Observe a tabela a seguir para ter uma idéia de quem são e para que server os operadores lógicos disponíveis.

Operador	Função
|	OR lógico
||	OR dinâmico
&	AND lógico
&&	AND dinâmico
^	XOR lógico
!	NOT unário lógico
|=	Atribuição de OR
&=	Atribuição de AND
^=	Atribuição de XOR
?:	if-then-else ternário

OPERADORES LÓGICOS

Dificilmente você precisará utilizar os operadores OR lógico e AND lógico. Tudo o que eles fazem pode ser feito com maior eficiência utilizando os operadores OR dinâmico e AND dinâmico.

Em expressões que utilizam o operador OR lógico, o resultado é true se ao menos um dos operandos possuir o valor true e será false somente se os dois operandos possuírem o valor false. Se o primeiro operando contiver o valor true, não importa o valor do segundo: o resultado será true. No entanto, com o operador OR lógico é realizada a avaliação do segundo operando mesmo que o primeiro contenha o valor true.

Para evitar o processamento desnecessário realizado com o operador OR lógico, é aconselhável o uso do operador OR dinâmico. Com ele, o resultado também é true se pelo menos um dos operandos contiver o valor true. Mas se o valor true é encontrado no primeiro operando, será descartada a avaliação do segundo.

O operador AND lógico é utilizado para avaliar dois operandos booleanos e o resultado é true somente se os dois contiverem o valor true. Caso um deles contenha o valor false, então o resultado será false, independente do que contenha o outro. Por dedução, se o primeiro operando contém o valor false, a avaliação do segundo operando é desnecessária. Mas com o operador AND lógico, o segundo operando é avaliado mesmo quando o primeiro contém o valor false.

O operador AND dinâmico pode ser usado par evitar o processamento desnecessário realizado com o operador AND lógico. Ele também é utilizado para formar expressões cujo resultado é true somente se os dois operandos contiverem o valor true. Mas, nesse case, se o valor false é encontrado no primeiro operando, a avaliação do segundo é descartada e o resultado será false.

Com o operador XOR lógico é possível comparar dois valores booleanos de modo a extrair um resultado true se, e somente se, um único operando contiver true. Isso significa que a expressão terá um resultado false em duas situações: se os dois operandos contiverem false ou se os dois contiverem true. Haverá um resultado true se um dos operandos contiver true e o outro contiver false.

O operador NOT unário lógico serve para inverter o estado de uma variável, expressão ou valor literal booleano. Ele é aplicado sobre um único operando e figura antes dele, invertendo o seu estado.

Os três operadores lógicos de atribuição são utilizados para armazenar o resultado da comparação de dois operandos booleanos no primeiro deles. O operedor de atribuição de OR compara dois operandos e armazena true no primeiro operando sempre que ao menos um deles contenha o valor true. O operador de atribuição de AND compara dois operandos e armazena true no primeiro somente se os dois contiverem o valor true. O operador de atribuição de XOR comprar dois operandos e armazena true no primeiro se, e somente se, apenas um deles contiver o valor true.

O operador if-then-else ternário (?:) é uma espécie de estrutura de seleção composta de uma expressão e duas declarações. Observe a sua sintaxe:

```
<expressão> ? <declaração 1> : <declaração 2>
```

Deve-se utilizar uma expressão da qual resulte um valor booleano. Caso o resultado dela seja true, a primeira declaração será executada; case contrário, a segunda declaração é executada. Essas duas declarações devem retornar o mesmo tipo de dado, que será atribuído a uma variável. Veja um exemplo:

```
int it1 = 10, it2 = 5;
int it3 = it2 == 0 ? 0 : it1 / it2;
```

As variáveis inteiras it1 e it2 foram inicializadas, respectivamente, com os valores dez e cinco. O operador if-then-else ternário foi utilizado para decidir que valor deve ser atribuído à variável it3. A expressão faz uma comparação de igualdade entre a variável it2 e o valor literal zero, que retornará o valor booleano false. Sendo assim, a segunda declaração será executada, dividindo o valor de it1 pelo valor de it2. O resultado dessa divisão é atribuído à variável it3. Caso o valor de it2 fosse zero, então a primeira declaração seria executada e o valor zero seria atribuído à variável it3.

O próximo exemplo o ajuda a ampliar seu entendimento acerca dos operadores lógicos.

OperadoresLogicos.java

```
public class OperadoresLogicos {
	public static void main(String[] args) {
		boolean bl1, bl2;
		bl1 = false;
		bl2 = true;
		System.out.println("Valores das variáveis:");
		System.out.println("bl1: " + bl1);
		System.out.println("bl2: " + bl2);

		System.out.println("\nOperações lógicas:");
		System.out.println("bl1 || bl2:\t");
		System.out.println(bl1 || bl2);
		System.out.println("bl1 && bl2:\t");
		System.out.println(bl1 && bl2);
		System.out.println("bl1 ^ bl2:\t");
		System.out.println(bl1 ^ bl2);
		System.out.println("!bl1:\t\t");
		System.out.println(!bl1);
		System.out.println("!bl2:\t\t");
		System.out.println(!bl2);

		System.out.println("\nAtribuições:");
		bl1 |= bl2;
		System.out.println("bl1 |= bl2:\t");
		System.out.println("bl1 recebeu: " + bl1);

		bl1 &= bl2;
		System.out.println("bl1 &= bl2:\t");
		System.out.println("bl1 recebeu: " + bl1);

		bl1 ^= bl2;
		System.out.println("bl1 ^= bl2:\t");
		System.out.println("bl1 recebeu: " + bl1);

		System.out.println("\nOperador ternario:");
		int it1 = 2, it2;
		System.out.println("it1 = " + it1);
```

```
            int it2 == 0 ? 0 : 10 / it1;
            System.out.println("it2 recebeu: " + it2);
        }
}
```

Precedência

Para resolver expressões algébricas, é preciso obedecer a certa ordem. Primeiro devem ser calculadas as partes que estão delimitadas por parênteses, depois as partes delimitadas por colchetes e, por último, as partes delimitadas por chaves. Na ausência desses separadores, realizam-se as multiplicações e as divisões e depois as adições e subtrações. Isso significa que existe uma precedência implícita entre as operações. Ela é utilizada sempre que não existem separadores indicando explicitamente a ordem a ser seguida para a resolução da expressão.

Esse mesmo raciocínio também é valido na linguagem Java. No entanto, a programação envolve um número maior de operações a serem controladas comparativamente à Álgebra. Em expressões que realizam duas ou mais operações, a ordem em que elas serão processadas depende da precedência de seus operadores e dos separadores presentes. A tabela a seguir contém a ordem de precedência dos principais operadores e separadores.

( )	[ ]	.	
++	- -	!	
*	/	%	
+	-		
>	>=	<	<=
=	!=		
&			
^			
|			
&&			
||			
?:			
=			

PRECEDÊNCIA DE OPERADORES E SEPARADORES

Cada uma das 13 linhas dessa tabela representa um nível de precedência. Oss três separadores que figuram na primeira linha (parênteses, colchetes, e ponto) são os elementos que têm maior precedência em uma expressão e todos aqueles que se encontram em uma mesma linha têm o mesmo nível de precedência. Observe as instruções a seguir:

```
int int1 = 5;
int it2 = 2 + 2 * 7 – 4 / --it1;
```

A primeira instrução declara a variável it1 e a inicializa com o valor cinco. Na segunda instrução, será atribuído à variável it2 o resultado de uma expressão que se compõe de cinco operações distintas: adição, multiplicação, subtração, divisão e decremento. Para saber como essa expressão será avaliada para extrair seu resultado, observe a ordem de precedência dos operadores na tabela anterior. Dos operadores presentes, aqueles que tem maior precedência é o operador de decremento. Depois, terão prioridade os operadores de divisão e de multiplicação. Por último, serão realizadas as operações de soma e subtração. Assim, o resultado dessa expressão será 15. Veja os passos que são dados para obter esse resultado:

- Obtém-se o valor decrementado de it1 (--it1 = 4)
- Multiplica-se 2 por 7 (2 * 7 = 14)
- Divide-se 4 por 4 (4 / 4 = 1)
- Soma-se 3 com 14 (2 + 14 = 16)
- Subtrai-se 1 de 16 (16 - 1= 15)

Para que as operações de adição e de subtração sejam realizadas antes das operações de multiplicação e de divisão, basta delimitá-las com parênteses. Veja como isso pode ser feito:

```
int it1 = 5;
int it2 = (2 + 2) * (7 – 4) / --it1;
```

Agora as operações de soma (2 + 2) e de subtração (7 – 4) serão realizadas antes mesmo da operação de decremento, uma vez que os parênteses denotam uma precedência de nível superior à de qualquer operador. O resultado, dessa vez, será três. Veja como o resultado será obtido:

- Soma-se 2 com (2 + 2 = 4)
- Subtrai-se 4 de 7 (7 - 4 = 3).
- Obtém-se o valor decrementado de it1 (-- it1 = 4).
- Multiplica-se 4 e 3 (4 * 3 = 12).
- Divide-se 12 por 4 (12 / 4 = 3).

Desse modo, sempre que desejar que uma operação seja realizada antes de outra de maior precedência, delimite-a com parênteses. Com este separador, você poderá determinar qualquer ordem para a execução das diversas operações que compõem uma mesma expressão.

Estruturas de Decisão

As estruturas de decisão são utilizadas para controlar o fluxo de execução dos programas, possibilitando que a leitura das instruções siga caminhos alternativos em função do estado de dados booleanos. Com elas, é possível condicionar a leitura de uma instrução ou de um bloco delas a uma ou mais condições que precisam ser satisfeitas.

Até aqui, todos os aplicativos de exemplo executam suas instruções de forma linear, ou seja, todas elas são lidas seqüencialmente, na ordem em que foram escritas no código. Com o uso de estruturas de decisão isso irá mudar e alguns trechos dos programas somente serão executados sob determinadas condições.

Há três estruturas de decisão disponíveis na linguagem Java: a estrutura if, a estrutura if-else e a estrutura switch-case. Cada uma delas possui um propósito distinto e serão analisadas separadamente.

Estrutura if

A estrutura de decisão if é utilizada para impor uma ou mais condições que deverão ser satisfeitas para a execução de uma instrução ou bloco de instruções. A sua forma geral é a seguinte:

```
if(<condição>) <instrução ou bloco>
```

A condição sempre irá figurar entre parênteses após a palavra reservada if e deve ser uma expressão booleana que resulte em um valor true ou false. A instrução ou o bloco de instruções somente serão executados caso o resultado dessa expressão seja true. Caso o resultado seja false, o fluxo de execução será desviado e a instrução ou o bloco não serão executados.

Havendo uma única instrução condicionada pela estrutura if, ela figura logo após a condição e termina com um ponto-e-vírgula. A sintaxe é a seguinte:

```
if(<condição>) <instrução>;
```

O programa Saida.java listado abaixo demonstra o uso da estrutura if para condicionar a execução de uma única instrução. Nesse exemplo, ela é utilizada para decidir se o aplicativo deve ou não ser encerrado antes que as instruções finais sejam executadas.

Saida.java

```
import javax.swing.JOptionPane;

public class Saida {
	public static void main(String[] args) {
		String st = “Informe seu nome: ”;
		st = JOptionPane.showInputDialog(st);
		if(st == null) System.exit(0);
		if(st.length() == 0) System.exit(0);
		st = “Nome informado: ” + st;
		JOptionPane.showMessageDialog(null, st, “Messagem”, 1);
	}
}
```

Observe que as duas estruturas if utilizadas no exemplo são compostas de uma única condição e de uma única instrução. A instrução é executada se a condição resultar em um valor booleano true. Mas se houver um bloco de instruções que deva ser executado quando a condição for verdadeira, é preciso delimitá-lo com o uso de chaves. A sintaxe da estrutura if, nesse case, é a seguinte:

```
if(<condição>) {
	<instrução 1>;
	<instrução 2>;
	...
	<instrução 3>;
}
```

Para entender como essa forma será aplicada, na prática, implemente e execute o programa Resposta.java. Esse exemplo é uma variação do anterior, no qual você receberá uma mensagem de alerta se cancelar o diálogo de entrada de dados e receberá uma mensagem de erro se confirmá-lo sem informar nada.

Resposta.java

```
import javax.swing.JOptionPane;

public class Resposta {
	public static void main(String[] args) {
		String st = “Informe seu nome: ”;
		st = JOptionPane.showInputDialog(st);

		if(st == null) {
			st = “Você não deve cancelar esse diálogo!”;
```

```
                JOptionPane.showMessageDialog(null, st, "Erro", 2);
                System.exit(0);
            }

            if(st.length() == 0) {
                st = "Você precisa informar seu nome!";
                JOptionPane.showMessageDialog(null, st, "Erro", 0);
                System.exit(0);
            }
            st = "Nome informado: " + st;
            JOptionPane.showMessageDialog(null, st, "Messagem", 1);
        }
}
```

Estrutura if-else

A estrutura de decisão if-else é uma variação da estrutura if. Ela é utilizada para impor uma ou mais condições que deverão ser satisfeiras para a execução de uma instrução ou bloco de instruções e possibilita a definição de uma instrução ou bloco de instruções a serem executados caso as condições não sejam satisfeitas. A sua forma geral é a seguinte:

```
if(<condição>) <instrução ou bloco>
else <instrução ou bloco>
```

A condição sempre irá figurar entre parênteses após a palavra reservada if e deve ser uma expressão booleana que resulte em um valor true ou false. A primeira instrução ou bloco de instruções somente serão executados caso o resultado dessa expressão seja true. Caso o resultado seja false, o fluxo de execução será desviado e a instrução ou o bloco posteriores ao else serão executados.

Implemente, compile e execute o programa Resultado.java para compreender melhor como funciona a estrutura de decisão if-else.

Resultado.java

```
import javax.swing.JOptionPane;

public class Resultado {
    public static void main(String[] args) {
        int nota1 = 0, nota2 = 0, nota3 = 0, media;

        String st = "Informe sua primeira nota (entre 0 e 10): ";
        st = JOptionPane.showInputDialog(st);
```

```
            if(st == null || st.length() == 0 ) {
                  st = "É preciso informar sua nota!";
                  JOptionPane.showMessageDialog(null, st, "Erro", 0);
                  System.exit(0);
            } else {
                  nota1 = Integer.parseInt(st);
            }

            st = "Informe sua segunda nota (entre 0 e 10): ";
            st = JOptionPane.showMessageDialog(null, st);

            if(st == null || st.length() == 0 ) {
                  st = "É preciso informar sua nota!";
                  JOptionPane.showMessageDialog(null, st, "Erro", 0);
                  System.exit(0);
            } else {
                  nota2 = Integer.parseInt(st);
            }

            st = "Informe sua terceira nota (entre 0 e 10): ";
            st = JOptionPane.showMessageDialog(null, st);

            if(st == null || st.length() == 0 ) {
                  st = "É preciso informar sua nota!";
                  JOptionPane.showMessageDialog(null, st, "Erro", 0);
                  System.exit(0);
            } else {
                  nota3 = Integer.parseInt(st);
            }

            media = (nota1 + nota2 + nota3) / 3;

            if(media >= 60) st = "Parabéns: você foi aprovado!";
            else st = "Sinto muito: você foi reprovado.";

            JOptionPane.showMessageDialog(null, st, "Messagem", 1);
      }
}
```

Para complementar a aprendizagem acerca do uso da estrutura if-else, implemente o aplicativo Media.java.

Media.java

```
import javax.swing.JOptionPane;

public class Media {
	public static void main(String[] args) {
		int nota1 = 0, nota2 = 0, media;

		String st = "Informe sua primeira nota (entre 0 e 10): ";
		st = JOptionPane.showInputDialog(st);

		if(st == null || st.length() == 0 ) {
			st = "É preciso informar sua nota!";
			JOptionPane.showMessageDialog(null, st, "Erro", 0);
			System.exit(0);
		} else {
			nota1 = Integer.parseInt(st);
		}

		st = "Informe sua segunda nota (entre 0 e 10): ";
		st = JOptionPane.showMessageDialog(null, st);

		if(st == null || st.length() == 0 ) {
			st = "É preciso informar sua nota!";
			JOptionPane.showMessageDialog(null, st, "Erro", 0);
			System.exit(0);
		} else {
			nota2 = Integer.parseInt(st);
		}

		media = (nota1 + nota2) / 2;

		if(media < 60) st = "insuficiente";
		else if(media < 80) st = "satisfatória";
		else if(media < 90) st = "boa";
		else st = "excelente";

		st = "Sua média foi " + st + "!\nMédia obtida: " + media;
		JOptionPane.showMessageDialog(null, st, "Messagem", 1);
	}
}
```

Esse exemplo realiza uma operação semelhante à realizada pelo programa Resultado.java. No entanto, ao invés de simplesmente informar se você foi aprovado ou reprovado, ele irá informar a média obtida e um conceito para a mesma, que poderá ser: insuficiente, satisfatória, boa ou excelente.

Estrutura switch-case

A estrutura de decisão switch-case, ou simplesmente switch, é uma forma simples para se definir diversos desvios no código a partir de uma única variável ou expressão. Havendo uma variável com diversos valores possíveis e sendo necessário um tratamento específico para cada um deles, o uso da estrutura if-else se torna confuso e dificulta a leitura do código. Nesse caso, a clareza e a facilidade estão do lado da estrutura switch.

A sintaxe geral da estrutura switch é a seguinte:

```
switch(<expressão ou variável>) {
      case <valor 1>:
            <instrução 1>;
            break;
      case <valor 2>:
            <instrução 2>;
            break;
      case <valor N>:
            <instrução N>;
            break;
      default:
            <instrução Default>;
}
```

Se for utilizada uma expressão, ela deve retornar um tipo de dado compatível com todos os valores especificados através das declarações case. Por dedução, todas as declarações case devem conter valores de um mesmo tipo. Caso seja utilizada uma variável, seu tipo também deve ser compatível com os valores das declarações case.

Cada um dos valores especificados com a declaração case deve ser um valor literal exclusivo. Se houver algum valor duplicado, será gerado um erro no momento em que você tentar compilar seu código.

A execução de uma estrutura switch-case é bastante simples. O valor da variável ou o valor de retorno da expressão é comparado com cada um dos valores contidos nas declarações case. Quando for encontrado um valor equivalente, as instruções que se encontram após a declaração case são executadas. Caso nenhum valor igual seja encontrado, as instruções contidas após a declaração detault é que serão executadas. No entanto, a declaração default é opcional; se ela não estiver presente e não houver nenhum valor nas declarações case que correspondam ao valor da variável ou expressão, então nenhuma instrução será executada.

A palavra reservada break é utilizada na estrutura switch para promover um desvio da execução para a linha posterior ao final de seu bloco. Geralmente, ele é utilizado com a última instrução de cada declaração case. Seu uso não é obrigatório; mas se for dedicado de lado, além do trabalho desnecessário de comparação dos valores de todos os cases posteriores com a expressão ou variável, as instruções da declaração default também acabarão sendo executadas.

É importante que você dê atenção especial às declarações break em estruturas switch. O compilador não irá avisá-lo se você esquecer um break. Um erro dessa natureza somente será detectado em testes realizados em tempo de execução, quando você perceber que seu programa não produz o resultado que deveria.

Para compreender melhor em que circunstância a estrutura switch deve ser aplicada e sanar as dúvidas acerca de sua sintaxe, resta implementar o exemplo prático que segue.

Meses.java

```
import javax.swing.JOptionPane;

public class Meses {
	public static void main(String[] args) {
		String st = “Informe um número entre 1 e 12”;
		st = JOptionPane.showInputDialog(st);
		int mes = Integer.parseInt(st);
		switch(mes) {
		case 1:
			st = “janeiro”;
			break;
		case 2:
			st = “fevereiro”;
			break;
		case 3:
			st = “março”;
```

```
                    break;
                case 4:
                    st = “abril”;
                    break;
                case 5:
                    st = “maio”;
                    break;
                case 6:
                    st = “junho”;
                    break;
                case 7:
                    st = “julho”;
                    break;
                case 8:
                    st = “agosto”;
                    break;
                case 9:
                    st = “setembro”;
                    break;
                case 10:
                    st = “outubro”;
                    break;
                case 11:
                    st = “novembro”;
                    break;
                case 12:
                    st = “dezembro”;
                    break;
                default:
                    st = “Mês inválido!”;
                    JOptionPane.showMessageDialog(null, st, “Erro”, 0);
                    System.exit(0);
                }
                st = “Você escolheu o mês de “ + st;
                JOptionPane.showMessageDialog(null, st, “Messagem”, 1);
                System.exit(0);
        }
}
```

Esse exemplo é bastante simplório, mas é adequado para que você fixe as noções fundamentais acerca do uso de estruturas switch. Quando ele for executado, irá produzir um diálogo onde você deverá informar um número à sua escolha e, em seguida, exibe uma mensagem de informação contendo o nome do mês que corresponde a esse número.

Estruturas de Repetição

As estruturas de repetição, também conhecidas como laços (loops), são utilizadas para executar repetidamente uma situação ou bloco de instruções enquanto determinada condição estiver sendo satisfeita.

Qualquer que seja a estrutura de repetição, ela conterá quatro elementos fundamentais: inicialização, condição, corpo e interação. A inicialização compõe-se de todo código que determina a condição inicial dos elementos envolvidos no laço. A condição é uma expressão booleana avaliada após cada leitura do corpo e determina se uma nova leitura deve ser feita ou se o não deve ser encerrado. O corpo compõe-se de todas as instruções que são executadas repetidamente. A interação é a instrução que deve ser executada depois do corpo e antes de uma nova repetição.

Há três tipos distintos de laço em Java e cada um deles é útil para realizar tipos distintos de repetições. São eles: while, do-while e for. Todos possuem os elementos supracitados, mas a sintaxe e a função de cada um são específicas.

Estrutura while

O laço while é a estrutura de repetição fundamental de Java. Ele é utilizado para executar repetidamente uma única instrução ou um bloco delas enquanto uma expressão booleana for verdadeira.

A sintaxe do laço while é a seguinte:

```
<inicialização>
while(<condição>) {
      <corpo>
      <iteração>
}
```

Veja que a inicialização precede o início do laço, Isso significa que você deve definir o estado inicial dos elementos que serão nele utilizados antes de seu cabeçalho. A palavra reservada while sempre será seguida de um par de parênteses, que delimitam a condição do laço. Essa condição deve ser uma expressão booleana e enquanto ela retornar true o laço continuará repetindo a execução das instruções contidas no corpo. O corpo pode ser tanto uma única instrução como um conjunto delas. Caso a iteração e o corpo sejam instruções distintas, então elas devem estar delimitadas por um par de chaves, formando um bloco. Do mesmo modo, quando o corpo contém mais de uma instrução, é preciso delimitá-lo por um par de chaves. A iteração deve ser uma instrução que modifica o estado de algum elemento utilizado na condição, de modo a garantir que ela retorne false em algum momento e encerre o laço.

O programa While.java contém um exemplo que demonstra o uso do laço while tanto para repetir uma única instrução quanto para repetir um bloco de instruções.

While.java

```
public class While {
	public static void main(String[] args) {
		System.out.println(“Primeiro while:”);

		int it = 0;
		while(it < 5) {
			System.out.println(++it); // iteração
		}

		System.out.println(“\nSegundo while:”);

		it = 69;
		while(it >= 65) {
			String st = “O número ” + it + “ equivale ao caractere ”;
			st = st + (char)it;
			System.out.println(st);
			it--; // iteração
		}
	}
}
```

Estrutura do-while

A estrutura de repetição do-while é uma variação da estrutura while. Existe uma diferença sutil, porém importante, entre elas. Em um laço while, a condição é testada antes da primeira execução das instruções que compõem seu corpo. Desse modo, se a condição for falsa na primeira vez em que for avaliada, as instruções desse laço não serão executadas nenhuma vez. Em um laço do-while, por outro lado, a condição somente é avaliada depois que seu bloco de instruções é executado pela primeira vez. Assim, mesmo que a condição desse laço seja false antes mesmo de ele iniciar, suas instruções serão executadas uma vez.

A sintaxe do laço do-while é a seguinte:

```
<inicialização>
do {
      <corpo>
      <iteração>
}while(<condição>);
```

Assim como em um laço while, a inicialização do laço do-while é seu primeiro elemento. Seu início efetivo é marcado pela palavra reservada do (faça) seguido das instruções que deverão ser repetidas e da instrução de iteração. Sua última linha sempre iniciará com o termo while (enquanto) seguido da condição entre parênteses. Veja que a condição somente será lida e avaliada depois que as instruções do corpo desse laço já tiverem sido executadas uma vez.

Como um exemplo prático, talvez algumas dúvidas remanescentes possam ser dissipadas. Implemente e execute o programa DoWhile.java. Ele contém um aplicativo que ilustra o uso dessa estrutura de repetição.

DoWhile.java

```
public class DoWhile {
      public static void main(String[] args) {
            System.out.println(“Primeiro do-while:”);

            int it = 9;
            do
                  System.out.println(++it); // iteração
            while(it < 5);

            System.out.println(“”);
            System.out.println(“Segundo do-while:”);
```

```
            it = 69;
            do {
                  String st = “O número ” + it + “ equivale ao caractere ”;
                  st = st + (char)it;
                  System.out.println(st);
                  it--; // iteração
            } while(it >= 65);
      }
}
```

Estrutura for

O laço for é uma estrutura de repetição compacta. Seus elementos de inicialização, condição e iteração são reunidos em forma de um cabeçalho e o corpo é disposto em seguida.

Veja a sintaxe de uma estrutura for:

```
for(<inicialização>;<condição>;<iteração>) {
      <corpo>
}
```

Observe que inicialização, condição e iteração aparecem entre parênteses após a palavra reservada for e separados por ponto-e-vírgula. A instrução ou bloco de instruções que ele repetirá são transcritos a partir da linha seguinte.

Na verdade, os laços for e while são apenas formas diferentes de uma mesma estrutura básica de repetição. Qualquer laço for pode ser transcrito em termos de um laço while e vice-versa. Do mesmo modo que em um laço while, se a condição de um laço for retornar false logo na primeira avaliação que for feita dela, então as instruções contidas em seu corpo jamais serão executadas.

Tradicionalmente, o laço for utiliza uma variável inteira para controlar o número de vezes em que deve repetir a execução das instruções. Essa variável é inicializada com um valor qualquer e é incrementado ou decrementado a cada interação. A condição geralmente verifica se essa variável já chegou em determinado valor para decidir se o laço deve ser encerrado.

No entanto, existem muitas outras formas de se utilizar o laço for. No exemplo contido no programa For.java há dois laços for, cada um deles sob uma forma distinta. Implemente e execute esse exemplo para fixar o entendimento.

For.java

```
import javax.swing.JOptionPane;

public class For {
	public static void main(String[] args) {
		String st = “Valores de it: ”;

		for(int it = 1;it <= 5;it++) {
			st = st + it + “ “;
		}

		st = JOptionPane.showInputDialog(st);
		for(st = ““;st == null || st.equals(““);st = st) {
			st = “Inforne seu nome”;
			st = JOptionPane.showInputDialog(st);
		}

		st = “Nome captado: “ + st;
		JOptionPane.showMessageDialog(null, st);
	}
}
```

Quebras de Laço

As quebras de laço são utilizadas para interromper o fluxo normal das estruturas de repetição while, do-while e for. Há dois tipos distintos de quebras de laço, representadas pelas palavras reservadas break e continue.

Há situações em que é preciso interromper um laço antes que sua condição se torne false. É para isso que serve o break. Figurando dentro do bloco de instruções de um laço qualquer, essa instrução encerra a estrutura de repetição, desviando a execução do código para alinha seguinte ao final desse laço.

Antes de passar para a análise da instrução continue, implemente e execute o programa Break.java para compreender melhor o uso da instrução break.

Break.java

```
import javax.swing.JOptionPane;

public class Break {
	public static void main(String[] args) {
		String st;
		while(true) {
```

```
                st = “Informe seu nome”;
                st = JOptionPane.showInputDialog(st);

                if(st == null) System.exit(0);
                if(!st.equals(“”)) break;
            }

            st = “Nome captado: “ + st;
            JOptionPane.showMessageDialog(null, st);
      }
}
```

Esse exemplo é bastante simples, mas deve ser suficiente para ilustrar a função da instrução break. Sua idéia central é criar um laço while infinito que solicite seu nome e encerrá-lo com break quando algo for informado.

Enquanto o break é utilizado para encerrar um laço, continue é utilizada para realizar uma nova repetição sem que a repetição anterior execute todas as suas instruções. Em laços while e do-while, uma instrução continue desvia o fluxo de execução para a iteração e, em seguida, a condição é lida novamente.

No exemplo contido no programa Continue.java a instrução continue foi utilizada para impedir que as últimas instruções de um laço for sejam lidas sem que você tenha informado seu nome. Implemente esse exemplo e execute-o.

Continue.java

```
import javax.swing.JOptionPane;

public class Continue {
      public static void main(String[] args) {
            for(int i = 0;true;i++) {
                String st = “Informe seu nome”;
                st = JOptionPane.showInputDialog(st);

                if(st == null) System.exit(0);
                if(st.equals(“”)) continue;

                if(i > 0) {
                      st = “Você confirmou“ + i +
                            “ vezes sem digitar nada.” +
                            “\nMas, ao final, informou seu nome, ” + st;
                } else {
```

```
                    st = “Obrigado por informar seu nome, “ + st;
                }

                JOptionPane.showMessageDialog(null, st);
            }
        }
}
```

Arrays

Os arrays são estruturas de dados que podem armazenar mais que um valor de um mesmo tipo de dado. Enquanto uma variável somente consegue armazenar um único valor, um array pode armazenar um conjunto de valores.

Em Java, os arrays são objetos. Isso significa que eles não se comportam como as variáveis e sim como instâncias de classes. Por isso, eles precisam ser declarados, instanciados (criados) e inicializados.

Existem dois tipos de arrays, a sabes: os vetores e as matrizes. Os vetores são arrays unidimensionais e as matrizes são arrays multi-dimensionais. Os tópicos seguintes irão explicar a sua função e forma de uso.

Vetores

Sendo arrays unidimensionais, os vetores funcionam como arranjos de valores de um mesmo tipo de dado. Você pode imaginar um vetor como uma linha de uma tabela, composta por diversas células. Em cada célula, é possível armazenar um valor.

A tabela abaixo representa um vetor, chamado vt, que contém quatro posições. Em cada posição está contido um número inteiro. Trata-se, portanto, de um vetor de números inteiros. Lembre-se que os vetores somente armazenam valores de um mesmo tipo de dado.

vt[0]	vt[1]	vt[2]	vt[3]
15	6	8	20

UM VETOR COM QUATRO POSIÇÕES

Observe que a primeira posição desse vetor é referenciada pelo número zero, entre colchetes, depois do seu nome: vt[0]. O número que aparece entre os colchetes é o índice utilizado para se fazer referência a uma posição específica do vetor. Isso significa que o primeiro valor é o seu elemento zero, o segundo valor é o elemento um, e assim sucessivamente. Você precisará utilizar essa forma de referência tanto para armazenar um valor em uma posição específica de um vetor quanto para recuperar esse valor.

Na prática, você precisa saber como realizar quatro operações básicas com vetores: declará-los, instanciá-los, inicializa-los e construí-los. A seguir, será explicada cada uma dessas operações.

A declaração de vetores é similar à declaração de variáveis. Ela pode ser feita de duas formas distintas. O resultado é o mesmo em ambos os casos. Você pode escolher uma delas em função de sua preferência pessoal, mas é interessante adotar um padrão a ser seguido. Veja as duas formas de se declarar um vetor:

```
<tipo>[] <nome;
<tipo> <nome>[];
```

Observe que a única diferença entre elas é a posição do par de colchetes. Na primeira forma, ele é disposto logo após o tipo e, na segunda, ele figura depois do nome do vetor. Em ambos os casos, a declaração do vetor inicia-se com a identificação do tipo de dado que ele irá armazenar, contém um nome e termina com ponto-e-vírgula. O tipo de dado escolhido determinará o tipo de valores que serão armazenados em todas as posições desse vetor e o nome será utilizado como identificador para se fazer referência às propriedades do vetor ou a uma de suas posições.

Veja dois exemplos de declaração de vetores:

```
int[] it;
char ch[];
```

O primeiro é um vetor de números inteiros, chamado it; o segundo é um vetor de caracteres, chamado ch. Isso significa que somente poderão ser armazenados números inteiros no vetor it e caracteres no vetor ch.

Depois de declarar um vetor, você precisa instanciá-lo. A instanciação é o processo pelo qual você aloca um endereço de memória para um objeto. Como fora dito antes, todos os arrays são tratados como objetos em Java. Por isso, ao contrário das variáveis, é preciso instanciá-los antes de utilizá-los. A instanciação pode ser entendida como a criação do vetor. Instanciar um vetor significa lhe atribuir um endereço de memória onde ele possa armazenar seus valores. Assim, se você tentar armazenar um valor em um vetor antes de instanciá-lo, ele ainda não terá onde gravar esse dado e ocorrerá um erro.

A sintaxe para instanciação de vetores é a seguinte:

```
<nome> = new <tipo>[<posições>];
```

A instanciação de um vetor começa com o nome dado a ele em sua declaração. O sinal de atribuição (=) é utilizado para indicar que será atribuído a ele um endereço de memória. O operador new é utilizado na criação de todos os objetos e também faz parte da sintaxe da criação de vetores. Depois do operador new, deve-se indicar o tipo desse vetor, que deve ser o mesmo utilizado em sua declaração. Em seguida, o número de posições que esse vetor conterá deve figurar entre colchetes. Trata-se do número de valores que ele poderá armazenar e deve ser representado por um número inteiro.

Veja como poderiam ser instanciados os vetores it e ch, declarados anteriormente:

```
it = new int[4];
ch = new char[3];
```

O vetor if foi instanciado com quatro posições e o vetor ch foi instanciado com três posições. Isso significa que o vetor it terá capacidade de armazenar até quatro números inteiros e que o vetor ch poderá armazenar até três caracteres.

A declaração e a instanciação de um vetor também podem ser feitas em uma única instrução. A sintaxe para isso é a seguinte:

```
<tipo>[] <nome> = new <tipo>[<posições>];
<tipo> <nome>[] = new <tipo>[<posições>];
```

Veja que continuam sendo válidas as duas formas de declaração. Mas, ao invés de encerrar a declaração com ponto-e-vírgula, você insere o sinal de atribuição, repete o tipo do vetor e indica, entre colchetes, a quantidade de posições que ele deve conter. Apesar de ser mais simples declarar e instanciar um vetor em uma única instrução, há casos em que você precisará declarar um vetor em um ponto do código e instanciá-lo em outro, porque a quantidade de posições é determinada de forma dinâmica. Assim, é preciso conhecer as duas formas.

Veja como os vetores it e ch poderiam ser declarados e instanciados em uma única instrução:

```
int it[] = new int[4];
char[] ch = new char[3];
```

Depois de declarar e instanciar um vetor, você já pode realizar as duas últimas operações: inicialização e consulta. Assim como você declara uma variável para armazenar um valor a ser recuperado posteriormente, o sentido de ser de um vetor é receber um conjunto de valores a serem consultados em momento posterior. A inicialização de vetores representa, pois, a atribuição de um valor para cada posição do vetor. A consulta é simplesmente a recuperação de um valor armazenado em uma posição de vetor.

A inicialização de um vetor pode ser feita posição a posição após sua declaração e instanciação. A sintaxe a ser observada para realizar essa operação pe a seguinte:

```
<nome>[<posição>] = <valor>;
```

A instrução que atribui um valor a um vetor deve iniciar-se, sempre, com o nome desse vetor seguido da posição na qual se deseja armazenar, entre colchetes. Depois do sinal de atribuição, deve constar um valor literal compatível com o tipo do vetor ou uma expressão que resulte em um valor compatível.

Veja como poderia ser atribuído um valor para cada uma das posições dos vetores it e ch:

```
it[0] = 5;
it[1] = 8;
it[2] = 3;
it[3] = 9;
ch[0] = 'A';
ch[1] = 'B';
ch[2] = 'C';
```

Como resultado dessas instruções, o vetor it passa a conter os seguintes números inteiros: cinco, oito, três e nove. O vetor ch, por seu lado, passa a conter os caracteres A, B e C. Lembre-se sempre que a primeira posição de qualquer vetor terá o índice zero. Por isso, os primeiros valores dos vetores it e ch foram atribuídos às posições zero.

Agora resta aprender como recuperar os dados contidos em um vetor para encerrar o entendimento acerca do uso de vetores. De nada adiantaria saber declarar, instanciar e armazenar valores em um vetor se não houvesse a possibilidade de consultá-los posteriormente.

A recuperação de valores de um vetor é bastante simples. Basta você utilizar uma referência a um aposição específica do mesmo, observando a seguinte sintaxe:

```
<nome>[<posição>]
```

Por exemplo, se desejasse recuperar o valor contido na posição dois do vetor ir, bastaria utilizar a seguinte referência: it[2]. Isso retornaria o número três, contido nessa posição. Você poderia utilizar esse valor dentro de uma expressão ou atribuí-lo a uma variável.

Veja alguns exemplos de recuperação e uso de valores de vetores:

```
int it2 = it[3];
int it3 = (it[0] + it[1] + it[2]) / it[3];
System.out.println(“A posição ' do vetor ch contém: “ + ch[1]);
```

A primeira dessas três instruções recupera o valor contido na posição três do vetor it (o número nome) e o atribui à variável it2. A segunda soma os valores contidos nas três primeiras posições do vetor irt e divide esse resultado pelo valor contido na posição três, atribuindo-o à variável it3 e divide esse resultado pelo valor contido na posição três, atribuindo-o à variável it3. A terceira instrução simplesmente imprime o valor contido na posição um do vetor ch (B).

Há, ainda, um modo de declarar, instanciar e inicializar um vetor em uma única instrução. A sintaxe e ser observada para isso é a que segue:

```
<tipo>[] <nome> = {<valor 1>, <valor 2>, ..., <valor N>};
<tipo> <nome>[] = {<valor 1>, <valor 2>, ..., <valor N>};
```

Continuam sendo válidas as duas formas de declaração. Mas aqui o vetor será declarado e já lhe será atribuído um conjunto de valores. O vetor será instanciado com a quantidade de posições equivalente ao número de valores que lhe forem atribuídos. Esses valores devem ser delimitados por um par de chaves e devem ser separados por vírgula.

Veja como os vetores it e ch poderiam ter sido declarados, instanciados e inicializados em instruções únicas.

```
int[] it = {5, 8, 3, 9};
char ch[] = {'A', 'B', 'C'};
```

Para fixar seu entendimento acerca dos vetores, já é o momento de implementar um exemplo prático completo.

Vetor.java

```
public class Continue {
	public static void main(String[] args) {
		int[] it;
		it = new int[3];
		it[0] = 65;
		it[1] = 66;
		it[2] = 67;

		System.out.println(“Conteúdo do vetor it:”);
		System.out.println(“it[0] = ” + it[0] = “\t”);
		System.out.println(“it[1] = ” + it[1] = “\t”);
		System.out.println(“it[2] = ” + it[2] = “\n”);
		System.out.println(“Qtde. de posições: ” + it.length + “\n\n”);

		double[] db = new double[2];
		db[0] = 1.25;
		db[1] = 2.54;

		System.out.println(“Conteúdo do vetor db:”);
		System.out.println(“db[0] = ” + db[0] = “\t”);
		System.out.println(“db[1] = ” + db[1] = “\n”);
		System.out.println(“Qtde. de posições: ” + db.length + “\n\n”);

		char[] ch = {'X', 'Y', 'Z'};

		System.out.println(“Conteúdo do vetor ch:”);
		for(int i = 0;i < ch.length;i++)
			System.out.println(“ch[“ + i + ”] = ” + ch[i] = “\t”);

		System.out.println(“\nQtde. de posições: ” + ch.length + “\n\n”);
	}
}
```

Matrizes

As matrizes são arrays multi-dimensionais. Isso significa que elas podem ter duas ou mais dimensões. O uso mais comum de matrizes é ara construção de estruturas de dados bidimensionais. Nesse case, elas são usadas para armazenar valores em forma de linhas e colunas, tal como ocorre em uma tabela.

Enquanto um vetor equivale a uma linha de uma tabela, uma matriz bidimensional pode simulas uma tabela inteira. Na instanciação de uma matriz bidimensional, especifica- se a quantidade de linhas e de colunas que ela conterá. A quantidade de posições da mesma pode ser calculada multiplicando-se a quantidade de linhas pela quantidade de colunas.

A tabela a seguir contém uma estrutura de dados que poderia tranqüilamente ser armazenada em uma matriz bidimensional.

54	35	47	92
19	74	58	42
75	66	81	13

UMA MATRIZ COM TRÊS LINHAS E QUATRO COLUNAS

As mesmas operações que você aprendeu a realizar com vetores serão aplicáveis às matrizes, quais sejam: declaração, instanciação, inicialização e consulta. É isso que é preciso aprender para se trabalhar com matrizes.

A sintaxe para declaração de matrizes é a seguinte:

```
<tipo>[][] <nome>;
<tipo> <nome>[][];
```

Note que, agora, são utilizados dois pares de colchetes ao invés de um. É isso que distingue a declaração de uma matriz bidimensional da declaração de um vetor. Para declarar uma matriz bidimensional devem-se incluir dois pares de colchetes após a indicação do tipo ou do nome.

Veja dois exemplos de declaração de matrizes bidimensionais:

```
int[][] it;
char ch[][];
```

A primeira é uma matriz de inteiros, chamada ir; a segunda é uma matriz de caracteres, chamada de ch. A matriz de inteiros somente poderá ser usada para armazenar valores numéricos inteiros e a matriz sh só aceitará caracteres Unicode.

Tendo declarado uma matriz, é preciso instanciá-la, tal como é feito com vetores. A sintaxe da instanciação de uma matriz bidimensional é a que segue:

```
<nome> = new <tipo>[<linhas>][<colunas>];
```

Na instanciação de um vetor, era preciso informar apenas a quantidade de posições depois do seu tipo. Na instanciação de uma matriz bidimensional, por outro lado, é preciso informar tanto a quantidade de linhas quanto a quantidade de colunas que ela conterá.

Veja como as matrizes it e ch, supracitadas, poderiam ser instanciadas:

```
it = new int[3][4];
ch = new char[2][3];
```

A matriz it foi instanciada com três linhas e quatro colunas e a matriz ch foi instanciada com duas linhas e três colunas. Isso significa que a matriz it terá capacidade para armazenar 12 números inteiros (3x4 = 12) e que a matriz ch poderá armazenar até seis caracteres (2x3 = 6).

Uma matriz também pode ser declarada e inicializada em uma única instrução. Para isso, deve-se observar a seguinte sintaxe:

```
<tipo>[][] <nome> = new <tipo>[<linhas>][<colunas>];
<tipo> <nome>[][] = new <tipo>[<linhas>][<colunas>];
```

A única diferença entre as duas formas dispostas acima é a posição dos dois pares de colchetes. Como já fora dito antes, o resultado de ambas as formas é o mesmo. Entretanto, você deve adotar sempre a mesma para manter a uniformidade de seus códigos. Veja como as matrizes it e ch poderiam ter sido declaradas e instanciadas em instruções únicas:

```
int [][] it = new int[3][4];
char ch[][] = new char[2][3];
```

Tal como é feito com vetores, a inicialização de matrizes pode ser feita posição a posição. Sendo assim, a inicialização consistirá na atribuição de um valor para cada uma das posições que compõem a matriz. A atribuição de um valor para uma posição da matriz deve ser feita observando-se a seguinte sintaxe:

```
<nome>[<linha>][<coluna>] = <valor>;
```

Em um vetor, bastaria indicar a posição onde o valor deveria ser armazenado. Em uma matriz bidimensional, é preciso informar duas coordenadas: a linha e a coluna. São essas duas informações que definem uma posição específica dentro da matriz.

Veja como poderiam ser inicializadas as matrizes it e ch:

```
it[0][0] = 54;
it[0][1] = 35;
it[0][2] = 47;
it[0][3] = 92;
it[1][0] = 19;
it[1][1] = 74;
it[1][2] = 58;
it[1][3] = 42;
it[2][0] = 75;
it[2][1] = 66;
it[2][2] = 81;
it[2][3] = 13;
ch[0][0] = 'A';
ch[0][1] = 'B';
ch[0][2] = 'C';
ch[1][0] = 'D';
ch[1][1] = 'E';
ch[1][2] = 'F';
```

A matriz it recebeu os valores da Tabela descrita anteriormente, tal como eles se apresentam nela. A matriz ch, por sua vez, recebeu os valores A, B e C na primeira linhas e os valores D, E e F na segunda linha. Observe que tanto o índice da linha quanto o índice da coluna começam em zero e que o primeiro dos dois índices é a indicação da linha.

A consulta ou recuperação de valores de uma matriz é feita de forma semelhante àquela que se faz em um vetor. No entanto, para recuperar um valor de uma matriz é preciso indicar a sua posição específica, que é determinada pelo número da linha e da coluna. Observe a sintaxe utilizada para fazer a referência a um valor de uma matriz bidimensional:

```
<nome>[<linha>][<coluna>]
```

Assim como ocorre com vetores, os valores recuperados de matrizes podem ser utilizados diretamente em expressões e instruções ou atribuídos a uma variável. Veja alguns exemplos disso.

```
int it2 = it[1][2] / 2;
char ch2 = ch[0][1];
```

A primeira instrução recupera o valor contido na coluna 2 da linha 1 (o número 58) da matriz it e o divide por dois; o resultado dessa operação é atribuído à variável it2. A segunda instrução simplesmente recupera o caractere armazenado na coluna 1 da linha 0 da matriz ch e o atribui à variável ch2.

Também é possível realizar a declaração, instanciação e inicialização de matrizes em uma instrução única. Asintaxe a ser observada para isso é a que segue:

```
<tipo>[][] <nome> = {
      {<vlrLin0Col0>, <vlrLin0Col1>, ..., <vlrLin0ColN>]},
      {<vlrLin1Col0>, <vlrLin1Col1>, ..., <vlrLin1ColN>]},
      {<vlrLin2Col0>, <vlrLin2Col1>, ..., <vlrLin2ColN>]},
      ...
      {<vlrLinNCol0>, <vlrLinNCol1>, ..., <vlrLinNColN>]},
};
```

Observe que, além do par de chaves que envolve todos os valores que são atribuídos à matriz, há um novo par de chaves para delimitar cada uma de suas linhas. Os conjuntos de valores que comporão cada linha são separados por vírgula, assim como os valores de uma mesma linha.

Para facilitar seu entendimento, veja como as matrizes it e ch poderiam ter sido declaradas, instanciadas e inicializadas através de instruções únicas:

```
int[][] it = {{54,35,47,92}, {19,74,58,42}, {75,66,81,13}};
char ch[][] = {{'A', 'B', 'C'}, {'D', 'E', 'F'}};
```

Agora só resta a você implementar um exemplo prático para reforçar a aprendizagem acerca da matrizes.

Matriz.java

```
public class Matriz {
      public static void main(String[] args) {
            int[][] it;
            it = new int[2][3];
            it[0][0] = 0;
            it[0][1] = 1;
            it[0][2] = 2;
            it[1][0] = 10;
            it[1][1] = 11;
            it[1][2] = 12;

            System.out.println(“Conteúdo da matriz it:”);
```

```
            System.out.println("it[0][0] = " + it[0][0] = "\t");
            System.out.println("it[0][1] = " + it[0][1] = "\t");
            System.out.println("it[0][2] = " + it[0][2] = "\n");
            System.out.println("it[1][0] = " + it[1][0] = "\t");
            System.out.println("it[1][1] = " + it[1][1] = "\t");
            System.out.println("it[1][2] = " + it[1][2] = "\n");
            System.out.println("Posições: " + it.length + "x" + it[0].length);

            char[][] ch = {
                  {'A', 'B', 'C'},
                  {'a', 'b', 'c'},
                  {'1', '2', '3'}
            };

            System.out.println("\nConteúdo da matriz ch:");
            for(int i = 0;i < ch.length;i++) {
                  for(int j= 0;j < ch[i].length;j++) {
                        System.out.println("ch[" + i + "][" + j + "] = "+
                                          ch[i][j] = "\t\n");
                  }
            }

            System.out.println("Posições: " + ch.length + "x" + ch[0].length);
      }
}
```

Tratamento de Exceções

Exceções são condições anormais que podem surgir enquanto um programa estiver sendo executado. Elas ocorrem em função de vários motivos, mas podem ser resumidas no seguinte: falhas no projeto ou implementação e erros cometidos pelo usuário durante o uso do programa.

Saba-se que não há um ser humano sequer que seja infalível. Ao contrário, as pessoas costumam cometer muitos erros no decorrer de suas vidas. Ora, todos os programas são desenvolvidos por pessoas (analistas, projetistas, programadores, etc.), as próprias ferramentas de desenvolvimento são criadas por pessoas e quem utiliza os programas também são pessoas (usuários). Nesse contexto, por mais bem projetado que tenha sido um programa e por mais bem preparados que sejam os seus usuários, a probabilidade de ocorrerem erros durante seu desenvolvimento e operação é grande.

Para compreender melhor o que é uma exceção, imagine a seguinte situação: o programa abre um diálogo onde o usuário deve informar um número inteiro, mas ele informa um valor que contém letras (tal como “34R” ou 2U6”). Quando o programa tentar utilizar esse dado para realizar uma operação matemática, por exemplo, ocorrerá uma exceção: não será possível convertê-lo para inteiro e, então, não conseguirá concluir a tarefa.

Esse é um exemplo de erro cometido pelo usuário do programa e que resulta em uma exceção. Ao projetista e ao programador cabe prever esse tipo de situações e tratá-las. O tratamento de exceções consiste exatamente em prever situações anormais que podem ocorrer e implementar uma solução para a mesma. Essa solução é um caminho alternativo no código para que o problema seja resolvido sem deixar inconsistências e permitir que o programa continue sendo operado.

No caso em questão, o programa poderia ter utilizado uma técnica simples de tratamento de exceções para que o dado informado pelo usuário fosse validado antes de ser utilizado em quaisquer operações dentro do programa. Tendo informado um valor inválido, o usuário seria notificado através de uma mensagem de erro e orientado a informar novamente o valor.

Antes de iniciar a aprendizagem das técnicas de tratamento de exceção, é importante que você veja, na prática, como elas se manifestam caso não sejam tratadas. Para isso implemente o programa Excecao.java

Excecao.java

```
import javax.swing.JOptionPane;

public class Excecao {
	public static void main(String[] args) {
		String st = “Informe um número inteiro válido”;
		st = JOptionPane.showInputDialog(null, st, “Informe”, 3);

		int it1 = Integer.parseInt(st);
		int it2 = it1 * it1;

		st = “O dobro de “ + it1 + “ é “ + it2;
		JOptionPane.showMessageDialog(null, st, “Mensagem”, 1);
	}
}
```

Quando for executado, esse aplicativo irá solicitar que seja informado um número inteiro válido. Ao invés de informar um número, tal como é solicitado, experimente cancelar esse diálogo. Como não foi prevista e tratada essa situação, uma exceção será disparada e a execução do programa será paralisada. Algumas informações acerca da exceção ocorrida serão fornecidas no modo textual. A única coisa que você poderá fazer nessa circunstância é encerrar o aplicativo, pressionando a seguinte combinação de teclas: CTRL+C.

Note que há a indicação do tipo de exceção (NumberFormatException) e a posição exata da linha de código que provocou a exceção: Exception.java:10. Isso significa que, ao tentar executar a instrução contida na linha 10, ocorreu uma exceção do tipo NumberFormatException. Esse tipo de exceção ocorre sempre que está sendo esperado um número e é encontrado um valor que não é e nem pode ser convertido para um tipo numérico compatível. Observe a instrução que provocou a exceção:

```
int it1 = Integer.parseInt(st);
```

Ela simplesmente está tentando converter o texto contido em st para um tipo int, de modo a poder atribuí-lo à variável it. No entanto, se você cancelou o diálogo produzindo na linha 8, st conterá uma referência nula (null) e isso não pode ser convertido para um tipo inteiro. Do mesmo modo, se você informar um texto que não é um número inteiro válido naquele diálogo e confirmá-lo, ocorrerá a mesma exceção na linha 10, quando o programa tentar fazer sua conversão.

Tendo compreendido o que são exceções, resta dominar as técnicas que podem ser aplicadas para tratá-las. Para isso, será necessário entender o uso de cinco palavras reservadas: try, catch, finally, throw e throws. Os três primeiros são os mais comuns: o termo try é utilizado para demarcar um bloco de código que pode gerar algum tipo de exceção, o termo catch oferece um caminho alternativo a ser percorrido no caso de ocorrer efetivamente uma exceção e o termo finally delimita um bloco de código que será executado em quaisquer circunstâncias (ocorrendo ou não uma exceção). Os termos throw e throws, por sua vez, são utilizados para disparar exceções. A diferença entre eles é que o primeiro figura como uma instrução no código para gerar, propositalmente, uma exceção e o segundo é manipulado pelo sistema para lançar uma exceção ocorrida. O termo throws somente será utilizado diretamente por você quando for declarar um método que pode gerar uma exceção com a qual ele próprio não consegue lidar.

A sintaxe geral de tratamento de exceções é a seguinte:

```
try {
      <bloco protegido>
} catch(<tipo da exceção 1>) {
```

```
      <tratamento para exceção do tipo 1>
      [throw(<nome1>);]
} catch(<tipo da exceção 2>) {
      <tratamento para exceção do tipo 2>
      [throw(<nome2>);]
}
...
} catch(<tipo da exceção N>) {
      <tratamento para exceção do tipo N>
      [throw(<nomeN>);]
} finally {
      <bloco de finalização>
}
```

Se você sabe que algumas instruções poderão provocar alguma exceção, deve colocá-las dentro de um bloco precedido do termo try. Logo após esse bloco, você deve incluir um bloco de código para o tratamento de cada tipo de exceção que poderá ser gerada. Cada um dos blocos de tratamento deve ser precedido do termo catch, seguido do tipo da exceção de que ele trata e de um nome. Você pode escolher qualquer nome, mas o tipo de exceção deve ser o identificador de uma classe de exceções existente.

As classes de exceção existentes em Java estão dispostas em forma de uma árvore hierárquica. No topo encontra-se a classe Exception e abaixo dela encontra-se centenas de classes para representar cada tipo de exceção. Algumas serão utilizadas com freqüência e outras apenas em situações particulares.

O bloco finally é opcional e serve para delimitar um bloco de instruções que devam ser executadas independentemente do que ocorra no bloco try. Ocorrendo ou não uma exceção, as instruções do bloco finally são executadas. As instruções contidas no bloco catch, por outro lado, somente são executadas quando ocorrer uma exceção que corresponda ao tipo declarado.

Para entender melhor como aplicar cada um dos termos relativos ao tratamento de exceções, sua análise irá enfatizar a estrutura try-catch e o bloco finally. Eles constituem a base fundamental da manipulação de exceções em Java.

Estrutura try-catch

A estrutura try-catch é a base fundamental para o tratamento de exceções. Enquanto o termo try delimita um bloco de instruções que podem gerar exceções, várias declarações catch podem ser utilizadas para definir um tratamento adequado para cada um dos tipos de exceção que podem ser gerados.

Se você sabe que um bloco de instruções poderá gerar algum tipo de exceção mas não sabe o nome de classe que representa aquele tipo específico, pode utilizar a classe Exception para implementar um tratamento genérico. Implemente o programa Codigo.java para entender isso.

Codigo.java

```
public class Codigo {
	public static void main(String[] args) {
		try {
			int it1 = Integer.parseInt(args[0]);
			int it2 = Integer.parseInt(args[1]);
			int it3 = it1 / it2;

			System.out.println("Resultado: " + it3);
		} catch(Exception e) {
			System.out.println("Ocorreu uma exceção!");
		}
	}
}
```

Em síntese, um tratamento genérico para exceções pode ser feito colocando as instruções que podem gerar algum tipo de exceção em um bloco try e declarando-se um bloco catch para tratamento das exceções do tipo Exception. Como todas as exceções são do tipo Exception, as instruções desse bloco catch serão executadas sempre que for gerado qualquer tipo de exceção no bloco try.

Apesar de ser muito prática, essa estratégia não é recomendável. À medida que você for conhecendo melhor as classes de exceção que representam anormalidades específicas, você deve procurar implementar um tratamento mais preciso. O exemplo contido no programa Tratadores.java lhe dará uma idéia acerca disso.

Tratadores.java

```
public class Tratadores {
	public static void main(String[] args) {
		String st = "";

		try {
			int it1 = Integer.parseInt(args[0]);
			int it2 = Integer.parseInt(args[1]);
			int it3 = it1 / it2;

			st = "Resultado: " + it3 +
```

```
                        "\nOperação concluida com sucesso!";
                System.out.println(st);
            } catch(ArrayIndexOutOfBoundsException e) {
                st = "Informe dois argumentos ao executar essa classe." +
                        "\nModelo: java Tratadores <arg1> <arg2>";
                System.out.println(st);
            } catch(NumberFormatException e) {
                st = "Os dois argumentos devem ser números inteiros." +
                        "\nModelo: Tratadores 14 2";
                System.out.println(st);
            } catch(ArithmeticException e) {
                st = "O segundo argumento não deve ser zero." +
                        "\nEle será utilizado como divisor de uma divisão.";
                System.out.println(st);
            }
        }
}
```

No exemplo anterior, foram declarados três blocos catch porque havia três tipos diferentes de exceções que poderiam ser geradas pelas instruções contidas no bloco try. Pode ser declarada a quantidade de blocos catch que se fizer necessária para tratar todos os tipos de exceção que possam ser geradas. Assim, pode haver várias declarações catch para um bloco try. No entanto, um bloco catch sempre irá se referir a um único bloco try.

Para reforçar um pouco mais seu entendimento do uso da estrutura try-catch, implemente o programa Validador.java.

Validador.java

```
import javax.swing.JOptionPane;

public class Validador {
      public static void main(String[] args) {
            while(true) {
                String st = "Inforne um número inteiro válido";
                st = JOptionPane.showInputDialog(null, st, "Informe", 3);
                if(st == null) break;

                try {
                    int it = Integer.parseInt(st);
                    st = String.valueOf(it * 3);
                    st = "O triplo de " + it + " é " + st;
```

```
                        JOptionPane.showMessageDialog(null, st, "Message", 1);
                        break;
                  } catch(NumberFormatException e) {
                        st = "O número informado não é válido!\n" +
                              "Informe novamente.";
                        JOptionPane.showMessageDialog(null, st, "Erro", 0);
                  }
            }
      }
}
```

O funcionamento da estrutura try-catch é simples. Mas para utilizá-la com eficiência você precisa ir familiarizando-se com os principais tipos de exceções existentes.

Bloco finally

Enquanto as instruções de uma declaração catch somente são executadas quando uma exceção ocorre no bloco try ao qual ela está vinculada, um bloco finally contém instruções que serão executadas em quaisquer circunstâncias. Se não ocorrer nenhuma exceção durante a execução das instruções do bloco try, as instruções do bloco finally serão executadas. Se ocorrer uma exceção, essas instruções serão executadas do mesmo jeito.

Portanto, se você quer ter certeza de que alguma instrução seja executada após aquelas instruções protegidas pelo bloco try, coloque-as em um bloco finally. Para entender melhor essa questão, implemente o programa Finally.java.

Finally.java

```
import javax.swing.JOptionPane;

public class Finally {
      public static void main(String[] args) {
            String st = "Informe um número inteiro válido";
            st = JOptionPane.showInputDialog(null, st, "Informe", 3);
            if(st != null) {
                  try {
                        int it = Integer.parseInt(st);
                        st = String.valueOf(it * it);
                        st = it + " * " + it + " = " + st;
                        JOptionPane.showMessageDialog(null, st, "Message", 1);
                  } catch(Exception e) {
                        st = "Ocorreu uma Exceção!\n" +
                              "Tipo: " + e.getClass() + "\n" +
```

```
                        "Mensagem: " + e.getMessage();
                    JOptionPane.showMessageDialog(null, st, "Erro", 0);
                    e.printStackTrace();
                } finally {
                    st = "O aplicativo será encerrado";
                    JOptionPane.showMessageDialog(null, st, "Message", 1);
                }
            }
        }
}
```

As instruções contidas no bloco finally serão executadas independente de ter havido ou não uma exceção do bloco try. Assim, sempre será exibida a mensagem de encerramento e, em seguida, o aplicativo é encerrado.

Objetos e Classes

Introdução à Programação Orientada a Objetos

A POO é o paradigma de programação predominante dos dias atuais, tendo substituído as técnicas de programação baseadas em procedimentos, "estruturadas", que foram desenvolvidas nos anos 70. A linguagem Java é totalmente orientada a objetos e é impossível programá-la no estilo de procedimentos com o qual talvez você esteja bem acostumado.

Vamos começar com uma pergunta que, aparentemente, parece não ter nada a ver com programação. Como empresas como Compaq, Dell, Gateway, Micron Technologies e outras principais fabricantes de computadores pessoais tornaram-se tão grandes tão rapidamente? A maior parte das pessoas haverá de dizer provavelmente que elas fizeram bons computadores de um modo geral e que os venderam a um bom preço numa era em que a demanda por computadores pessoais havia disparado. Mas, vamos nos aprofundar mais um pouco e pensar, como elas foram capazes de fabricar tantos modelos tão rapidamente, respondendo às alterações que estavam acontecendo tão rápido?

Bem, uma grande parte da resposta está em que essas empresas redistribuíram um bocado de trabalho. Elas compravam componentes de fabricantes conhecidos e depois os montavam. Elas freqüentemente não investiam tempo nem dinheiro no projeto e na elaboração de fontes de alimentação, unidades de disco, placas mão e outros componentes. Isso tornou possível para essas empresas produzir um produto e fazer modificações rapidamente por menos custo do que se tivessem feito toda a engenharia por si mesmos.

O que os fabricantes de computadores pessoais estava comprando era "funcionalidade já pronta". Por exemplo, quando eles compravam uma fonte de alimentação, eles estavam comprando algo com certas propriedades (tamanho, forma, etc.) e uma certa funcionalidade (saída controlada de alimentação, energia disponível etc.). A (antiga) Compac deixou de projetar engenharia de todas as peças de suas máquinas e passou a comprar muitas partes, ela teve um aumento significativo nos lucros.

A POO nasceu da mesma idéia. Um programa é feito de objetos com certas propriedades e operações que os objetos podem realizar. O estado atual pode mudar com o passar do tempo, mas você vai depender sempre de que os objetos não interajam entre si de formas não documentadas. Se você vai elaborar ou comprar um objeto vai depender de orçamento ou de prazo. Porém, basicamente, enquanto os objetos atenderem às especificações esperadas, não há por que se preocupar em saber como a funcionalidade foi implementada. Na POO, você só se importa com o que o objeto expõe. Assim da mesma forma os fabricantes de cópias não se importam com o que está dentro de uma fonte de alimentação desde que ela faça o que querem. A maioria dos programadores Java não quer saber como o componente clipe de áudio, por exemplo, foi implementado, desde que faça o que é desejado.

A programação tradicional estruturada consiste em projetar um conjunto de funções para resolver um problema.(Essas funções são geralmente chamadas de algoritmos). O passo seguinte convencional é encontrar a forma apropriada de armazenar os dados. É por isso que o criador do Pascal original, Niklaus Wirth, chamou seu famoso livro de Algoritmos + Estruturas de Dados = Programas (Prentice Hall, 1975). Observe que no título do livro de Wirth, os algoritmos vinham antes e as estruturadas de dados depois. Isso imita a forma com que os programadores trabalhavam naqueles tempos. Primeiro, decidia-se como manipular os dados; depois, que estrutura impor aos dados a fim de tornar mais fácil o processamento. A POO inverte essa ordem e coloca as estruturas de dados em primeiro lugar, examinando depois os algoritmos que vão processar os dados.

A chave para ser mais produtivo em POO é tornar cada objeto responsável pela realização de um conjunto de tarefas relacionadas. Se um objeto depender de uma tarefa que não seja de sua responsabilidade, ele vai precisar ter acesso a um objeto cujas responsabilidades incluem essa tarefa. O primeiro objeto então pede ao segundo objeto para realizar a tarefa. Isso é feito com uma versão mais generalizada da chamada da função que você conhece na programação por procedimentos. (Lembre-se de que em Java essas chamadas de funções são geralmente chamadas de chamadas a métodos). No jargão da POO, há clientes que enviam mensagens a objetos servidores.

Em particular, um objeto nunca deveria manipular diretamente os dados internos de outro objeto. Toda a comunicação deve ser através de "mensagens", ou seja, chamadas a métodos. Ao projetar seus objetos para lidar com todas as mensagens apropriadas e manipular seus dados internamente, maximiza-se a reutilização, reduz-se a dependência da dados e minimiza-se o tempo de depuração.

Evidentemente, assim como ocorre com os módulos de uma linguagem orientada a procedimentos, não se quer que um objeto individual faça coisas demais de uma só vez. Tanto o projeto quanto a depuração são simplificados quando se formam objetos pequenos que realizam algumas poucas tarefas em vez de objetos enormes com dados internos extremamente complexos, com centenas de funções para manipular os dados.

O vocabulário da POO

Agora, é necessário entender a terminologia da POO para poder prosseguir. O termo mais importante é classe. Uma classe é geralmente descrita como o modelo ou a forma a partir da qual um objeto é criado. Isso leva ao modo padrão de pensar sobre as classes: como sendo as formas com as quais se fazem biscoitos. Os objetos são os biscoitos. A "massa do biscoito", na forma de memória, também terá de ser providenciada. A linguagem Java é ótima para esconder as etapas da "preparação da massa dos biscoitos". É só usar a palavra-chave "new" para obter memória e o sistema coleta de lixo (garbage collector) interno irá comer os biscoitos que ninguém mais quer. (Bem, nenhuma analogia é perfeita...). Quando se cria um objeto a partir de uma classe, diz-se que foi criada uma instância da classe. Quando se tem uma instrução como:

```
Date data = new Date();
```

Internamente o operador será usado para criar uma nova instância da classe "Date".

Tudo o que se escreve em Java está dentro de uma classe. A biblioteca Java padrão fornece centenas de classes para diversos propósitos tais como projeto de interface de usuário e programação em rede. Apesar de tudo, você ainda tem de criar suas próprias classes em Java para descrever os objetos dos domínios do problema de seus aplicativos e adaptar as classes que são fornecidas pela biblioteca padrão para atender a seus próprios objetivos.

Quando se começa a escrever classes em Java, outro princípio da POO facilita isso: as classes podem ser (e em Java sempre são) formadas a partir de outras classes. Nós dizemos que uma classe que é formada a partir de outra classe a estende. A linguagem Java, de fato, vem com uma espécie de "classe base", ou seja, uma classe a partir da qual todas as outras classes são elaboradas. Em Java, todas as classes estendem essa classe base chamada "Object".

Ao estender uma classe base, a nova classe inicialmente tem todas as propriedades e métodos de seu progenitor. Pode-se escolher então entre modificar ou simplesmente manter qualquer método da progenitora e também adicionar novos métodos que aplicam- se somente à classe filha. O conceito genérico de estender uma classe base é chamado de herança.

Encapsulamento (algumas vezes chamada de ocultamento de dados) é outro conceito-chave ao se trabalhar com objetos. Formalmente, o encapsulamento são é nada mais que combinar dados e comportamento em um pacote e ocultar a implementação dos dados do usuário do objeto. Os dados em um objeto são geralmente chamados de variáveis de instância ou campos; e as funções e procedimentos de uma classe Java são chamados métodos. Um objeto específico que é uma instância de uma classe terão valores específicos em seus campos que definem seu estado atual.

Nunca será demais enfatizar que a chave para se fazer o encapsulamento funcionar é fazer programas que nunca tenham acesso direto às variáveis de instância (campos) de uma classe. Os programas precisam interagir com seus dados somente através dos métodos do objeto. O encapsulamento é a forma de se das aos objetos seu comportamento de "caixa preta", característica-chave para a reutilização e a confiabilidade. Isso significa que um objeto pode mudar totalmente como ele armazena seus dados, mas, enquanto continuar usando os mesmos método para processar os dados, nenhum outro objeto vai saber disso ou se importar com isso.

Objetos

Para trabalhar com POO, deve-se poder identificar três características-chave dos objetos. (para aqueles que ainda se lembram, é análogo à forma do "Quem, O quê e Onde" que os professores ensinavam para caracterizar um evento). As Três perguntas- chave aqui são:

- Qual é o comportamento do objeto?
- Qual é o estado do objeto?
- Qual é a identidade do objeto?

Todos os objetos que são instâncias de uma mesma classe compartilham uma semelhança de família apresentando um comportamento semelhante. O comportamento de um objeto é definido pelas mensagens que ele aceita.

Em seguida, cada objeto guarda informações sobre o que ele é no momento e como chegou a isso. Isto é o que geralmente é chamado de estado do objeto. O estado de um objeto precisa resultar de mensagens enviadas a este objeto (caso contrário o encapsulamento seria desrespeitado).

Contudo, o estado de um objeto não descreve completamente o objeto, pois cada objeto rem uma identidade distinta. Por exemplo, em um sistema de processamento de encomendas, duas encomendas são distintas mesmo se pedirem a mesma coisa. Observe que os objetos individuais que são instâncias de uma classe sempre tem identidades diferentes e, geralmente, têm estados diferentes.

Essas características-chave podem influenciar umas às outras. Por exemplo, o estado de um objeto pode influenciar seu comportamento. (Se uma encomenda for "enviada" ou "paga" ela pode rejeitar uma mensagem que solicita o acréscimo ou a remoção de itens da encomenda. Da mesma forma, se um pedido estiver "vazio", ou seja, ainda nenhum item tiver sido solicitado, ele não poderá ser enviado).

Recomendação: uma regra simples na identificação de classes é procurar por substantivos na analise de um problema. Já os métodos, por outro lado, correspondem aos verbos.

Por exemplo, em um sistema de processamento de pedidos, alguns desses nomes substantivos são:

- item
- pedido (ou encomenda)
- endereço de remessa
- pagamento
- conta

Esses nomes podem levar às classes: *Item, Pedido* etc.

Em seguida, procuram-se os verbos. Itens são adicionados aos pedidos. Pedidos são enviados ou cancelados. Pagamentos são aplicados aos Pedidos. Com cada verbo, como "adicionar", "enviar", "cancelar" e "aplicar", é necessário identificar o objeto que tem a maior responsabilidade em efetuar essa ação. Por exemplo, quando um item novo é adicionado a um pedido, o objeto pedido deverá ser o encarregado, já que ele sabe como guardar e classificar os itens. Ou seja, *adicionar* deverá ser um método da classe *Pedido* que tem um objeto *Item* como parâmetro.

Evidentemente, a regra, "substantivo e verbo" é apenas uma diretriz e somente a experiência poderá ajudar você a decidir que substantivos e verbos são os mais importantes ao se elaborarem as classes.

Relação entre Classes

As relações mais comuns entre classes são:

- uso
- inclusão ("tem um")
- herança ("é um")

A relação uso é a mais óbvia e também a mais genérica. Por exemplo, a classe Pedido usa a classe Conta, já que os objetos Pedido precisam acessar os objetos Conta para verificar o crédito. Mas a classe Item não usa a classe Conta pois os objetos Item nunca precisam se preocupar com as contas dos clientes. Assim, uma classe usa outra classe se ela manipula objetos dessa classe.

De um modo geral, uma classe A usa uma classe B se:

- um método de A envia uma mensagem para um objeto da classe B, ou
- um método de A cria, recebe ou retorna objetos da classe B.

Recomendação: Tente minimizar o número de classes que usam umas às outras. O ponto é que, se uma classe A não sabe da existência de uma classe B, ela também fica indiferente ao que acontece com a classe B e isso significa que qualquer alteração na classe B não vai introduzir erros na classe A.

A relação inclusão é fácil de entender porque ela é concreta; por exemplo, um objeto Pedido contém objetos Item. A inclusão significa que objetos da classe A contêm objetos da classe B. Evidentemente, a inclusão é um caso especial de uso; se um objeto A contém um objeto B, então pelo menos um método da classe A irá usar esse objeto da classe B.

A relação de herança denota especialização. Por exemplo, uma classe PedidoExpresso será herdeira de uma classe Pedido. A classe especializada PedidoExpresso tem métodos especiais para lidar com prioridades e um método diferente para calcular as taxas de remessa, enquanto que os outros métodos, como adição de itens e cobrança serão simplesmente herdados da classe Encomenda. Em geral, se a classe A estender a classe B, a classe A vai herdar métodos da classe B e ter recursos a mais.

Recomendações para o Projeto de Classes

1. Sempre mantenha os dados privados
    - Esta regra vem antes de tudo: qualquer coisa diferente vai violar o encapsulamento. Você pode ter de escrever um método "acessador" (get) ou "modificador" (set) ocasionalmente, mas será ainda melhor manter privados os campos de instância.
2. Sempre inicialize os dados.
    - A linguagem Java não inicializa as variáveis locais automaticamente, mas inicializa as variáveis de instância de objetos. Não confie nos valores a priori, mas inicialize as variáveis explicitamente, seja fornecendo um valor inicialmente ou especificando os padrões em todos os construtores.

3. Não use tipos básicos em demasia numa classe.

   - A idéia é substituir os usos relacionados múltiplos de tipos básicos por outras classes. Isso mantém as classes mais fáceis de entender e alterar. Por exemplo, substitua os campos de instância a seguir em uma classe *Cliente* por uma nova chamada *Endereco*.

     ```
     private String rua;
     private String cidade;
     private String estado;
     private int cep;
     ```

   - Dessa forma, você poderá lidar facilmente com alterações de endereço, bem como com a necessidade de lidar com endereços internacionais.

4. Nem todos os campos precisam de "mecanismos de consulta" (get) e "mecanismos de atribuição" (set) de campos individuais.

   - Pode ser necessário obter e especificar o salário de alguém, o que é normal. Porém, você certamente não vai querer alterar sua data de contratação, uma vez o objeto construído. E com bastante freqüência, os objetos têm variáveis de instância que não se quer que outros leiam ou modifiquem.

5. Use uma forma padrão de definição das classes.

   - Nós sempre listamos o conteúdo das classes na seguinte ordem:

     1. recursos públicos
     2. recursos de escopo de pacote
     3. recursos privados
     4. constantes
     5. construtores
     6. métodos
     7. métodos estáticos

   - Use você essa ordem ou não, a coisa mais importante é ser consistente.

6. Divida classes com tarefas demais.

    - Esta recomendação é, evidentemente, um tanto quento vaga, pois "demais" é algo relativo. Contudo, se houver uma forma óbvia de tornar uma classe complicada em duas conceitualmente mais simples, aproveite a oportunidade. (Por outro lado, não exagere; 10 classes, cada uma com somente um método, é geralmente tiro de canhão em mosquito).

7. Dê nomes às classes e métodos que representem suas tarefas.

    - Assim como as variáveis devem ter nomes representativos, as classes também devem seguir essa metodologia.

    - Uma boa convenção é que um nome de classe deva ser um substantivo (Pedido) ou um substantivo junto com um adjetivo (ex.: *PedidoExpresso* ou *EnderecoCobranca*). Como nos métodos, siga a convenção padrão dos métodos "acessadores" que começam com *get* em minúsculas (*getDia*) e "modificadores" que começam com *set* também em minúsculas (*setSalario*).

Primeiros Passos com a Herança

Vamos supor que você trabalhe para uma empresa cujos diretores são tratados de maneira bem diferente que os outros empregados (classe Empregado). Seus aumentos são calculados diferentemente; eles têm direito a uma secretária e coisas assim. Esse é o tipo de situação que, em POO, pede herança. Por que? Bem, é necessário definir uma nova classe, (Gerente) e adicionar funcionalidade. Mas, pode-se reter algumas das coisas que já foram programadas na classe Empregado e todos os campos de instância da classe original podem ser preservados. De forma mais abstrata, há uma evidente relação entre Empregado e Gerente, pois cada gerente é um empregado: essa relação é a marca registrada da herança.

Eis o código da classe Empregado.

```
import java.util.*;

public class Empregado {
	private String nome;
	private double salario;
	private Date dataContratacao;

	public Empregado(String nome, double sal, Date data) {
		this.nome = nome;
		this.salario = sal;
```

```
            this.dataContratacao = data;
      }

      public void aumentaSalario(double porPercentual) {
            salario *= 1 + porPercentual / 100;
      }

      public int anoContratacao() {
            Calendar dt = Calendar.getInstance();
            dt.setTime(dataContratacao);
            return dt.get(Calendar.YEAR);
      }

      public String toString() {
            return String.format(new Locale("pt", "BR"), "%s %,.2f %d”,
                        nome, salario, anoContratacao());
      }
}
```

A palavra-chave extends na primeira linha da classe Gerente indica que está sendo criada uma nova classe que deriva de uma classe existente. A classe existente é chamada de superclasse, classe base ou classe progenitora. A nova classe é chamada de subclasse, classe derivada, ou classe filha. Os termos superclasse e subclasse são os mais usados habitualmente por programadores Java.

A classe Empregado é uma superclasse, mas não porque ela é superior a sua subclasse ou contenha mais funcionalidade. Na verdade, o que acontece é o oposto: as subclasses têm mais funcionalidades que suas superclasses. Por exemplo, como veremos ao analisar o restante do código da classe Gerente, essa classe encapsula mais dados e tem maior funcionalidade que sua superclasse Empregado.

Agora observe o construtor da classe Gerente:

```
public Gerente(String nome, double sal, Date data) {
      super(nome, sal, data);
      nomeSecretaria = "";
}
```

A palavra-chave super sempre se refere a uma superclasse (neste case, Empregado). De modo que a linha

```
super(nome, sal, data);
```

é um modo mais sucinto de chamar o construtor da classe Empregado usando nome, sal, e dia como parâmetros. O motivo dessa linha de código é que todo o construtor de uma subclasse necessita chamar um construtor para os campos de dados da superclasse. Se isto não ocorrer, o compilador Java irá procurar pelo construtor padrão da superclasse, que na sua ausência gera-rá um erro de compilação.

Se você comparar a classe Gerente com a classe Empregado, vai notar que muitos dos métodos da classe Empregado não são repetidos na classe Gerente. Isto porque, a menos que especificado, uma subclasse sempre usa os métodos da superclasse. Em particular, quando se elabora a subclasse por herança a partir da superclasse, basta indicas as diferenças entre a subclasse e a superclasse. A capacidade de reutilizar métodos da superclasse é automática.

Os Objetos sabem como fazer seu trabalho: Polimorfismo

É importante entender o que acontece quando a chamada de um método é aplicada a objetos de vários tipos em uma hierarquia de heranças. Lembre-se de que na POO são enviadas mensagens para objetos, pedindo-lhes que realizem certas ações. Ao enviar uma mensagem que pede para uma subclasse aplicar um método usando certos parâmetros, eis o que acontece:

- A subclasse verifica se ela tem ou não um método com esse nome e com exatamente os mesmos parâmetros. Se tiver, usa-o.

Caso não,

- a classe progenitor torna-se responsável pelo processamento da mensagem e procura por um método com esse nome e esses parâmetros. Se encontrar, chama esse método.

Como o processamento da mensagem pode continuar subindo pela seqüência de heranças, as classes progenitoras são verificadas até que a cadeia pára ou até que um método coincidente seja encontrado. (Se não houver nenhum método coincidente em nenhum lugar da seqüência de heranças, vai surgir um erro em tempo de compilação). Observe que podem existir métodos com mesmo nome na seqüência. Isso nos leva a uma das regras fundamentais de herança:

- Um método definido em uma subclasse com o mesmo nome e mesma lista de parâmetros que um método em uma de suas classes antecessoras oculta o método da classe ancestral a partir da subclasse.

A capacidade de um objeto em decidir que método aplicar a si mesmo, dependendo de onde ele está na hierarquia de heranças, é geralmente chamada de polimorfismo. A idéia por trás do polimorfismo é que, embora a mensagem possa ser a mesma, os objetos podem responder diferentemente. O polimorfismo pode ser aplicado a qualquer método que seja herdado de uma superclasse.

A chave para fazer o polimorfismo funcionar é chamada de ligação tardia (late binding). Isso significa que o compilador não gera código para chamar um método em tempo de compilação. Em vez disso, cada vez que se aplica um método a um objeto, o compilador gera código para calcular que método deve ser chamado, usando informações de tipo do objeto. (Esse processo é também chamado de ligação dinâmica ou despacho dinâmico). O mecanismo de chamada de método tradicional é chamado de ligação estática (static binding ou early binding), pois a operação a ser executada é totalmente determinada em tempo de compilação. A ligação estática depende apenas do tipo de variável objeto; já a ligação dinâmica depende do tipo do objeto real em tempo de execução.

Para resumir, herança e polimorfismo permitem ao aplicativo determinar a maneira geral de proceder. As classes individuais na hierarquia de heranças são responsáveis pelos detalhes – usando polimorfismo para determinar que métodos chamar. O polimorfismo em uma hierarquia de heranças é algumas vezes chamado de polimorfismo verdadeiro. A idéia é distingui-lo do tipo mais limitado de sobrecarga de nome (overloading) que não é resolvido dinamicamente, mas estaticamente em tempo de compilação.

Como evitar Herança: Classes e Métodos Finais (Final)

Ocasionalmente, pode-se querer impedir que uma classe seja derivada a partir de outras. As Classes que não podem ser classes-mãe são chamadas de classes finais e usa-se o modificador final na definição da classe para indicar isso.

Pode-se também fazer final um método específico de uma classe. Se isso for feito, então nenhuma subclasse poderá sobrepor ou substituir esse método. (Todos os métodos de uma classe final são automaticamente “finais”). Dois motivos para fazer uma classe ou um método serem finais:

1. Eficiência

- A ligação dinâmica é mais trabalhosa, em termos de processamento, que a ligação estática. Ou seja, os métodos virtuais executam mais devagar. O mecanismo de ligação dinâmica é um pouco menos eficiente que a chamada direta a um procedimento. Mais importante, o compilador não tem como substituir um método trivial por um código "in-line" porque é possível que uma classe derivada sobreponha esse código trivial. O compilador pode colocar métodos finais in-line. Por exemplo, se "e.getNome()" for final, o compilador poderá substituí-lo por "e.nome". (Assim, pode-se obter todos os benefícios do acesso direto a campos de instância ser violar o encapsulamento).

2. Segurança

- A flexibilidade do mecanismo de despacho dinâmico significa que não há controle do que acontece quando se chama um método. Ao enviar uma mensagem, como "e.getNome()", é possível que "e" seja um objeto de uma classe derivada que redefiniu o método "getNome" para retornar um string totalmente diferente. Ao fazer o método final, essa ambigüidade é evitada.

Conversão de Tipo Explícita (cast)

Para efetuar a conversão de um tipo primitivo para outro utilizamos em Java uma notação especial chamada de "cast". O exemplo abaixo, converte a expressão "x" em um inteiro, descartando a parte fracionária.

```
double x = 3.405;
int nx = (int)x;
```

Assim como você, ocasionalmente, precisa converter um número de ponto flutuante em inteiro, também pode precisar converter a referência de um objeto de uma classe para outra. Assim como na conversão dos tipos básicos, esse processo é chamado de "cast". Para realmente fazer uma conversão explícita, usa-se uma sintaxe semelhante à que foi usada para converter dados de variáveis de tipos básicos. Cerque o tipo do alvo com parênteses e coloque-o antes da referência de objeto que se quer converter. Por exemplo:

```
Gerente chefe = (Gerente)grupo[0];
```

Só há uma razão para que se queira fazer uma conversão de tipo explícita – usar um objeto em sua plena capacidade após seu tipo verdadeiro ter sido subestimado. Por exemplo, na classe "Gerente", o array "grupo" teria de ser um arrai de objetos "Empregado" já que alguns de seus itens era empregados normais. E teríamos de converte os elementos do array, de volta para "Gerente", a fim de acessar qualquer um de seus novos campos.

Como sabemos, em Java, toda variável objeto tem um tipo. O tipo descreve o gênero de objeto ao qual a variável faz referência e o que ela pode fazer. Por exemplo, "grupo[i]" refere-se a um objeto "Empregado" (e, portanto, também pode fazer referência a um objeto "Gerente")

O código é elaborado com base nessas descrições e o compilador verifica a correção dessas descrições de variáveis. Se um objeto de subclasse for atribuído a uma variável de superclasse, é como se você estivesse prometendo menos coisas e o compilador vai simplesmente permitir que isso seja feito. Mas, se você atribuir um objeto de superclasse a uma variável de subclasse, é como prometer mais do que havia sido especificado e o compilador exige confirmação disso através do uto de "cast" para o tipo da superclasse.

O que aconteceria se você tentasse casar para baixo na cadeia de herança e estivesse "mentindo" a respeito do que um objeto contém? Durante a execução do programa o sistema lançaria uma exceção e o programa iria, normalmente, ser interrompido. É uma boa prática de programação descobrir se um objeto é ou não uma instância de outra classe antes de fazer uma conversão de tipo explícita "cast". Isso é realizado com o operador "instanceof".

```
if(grupo[1] instanceof Gerente)
      chefe = (Gerente)grupo[1];
```

Finalmente, o compilador não vai permitir que seja feito um "cast" se não houver possibilidades de que a conversão seja bem sucedida.

```
Window w = (Window)grupo[1];
```

O Código acima gera um erro em tempo de compilação pois "Window" não é uma subclasse de "Empregado".

Resumindo:

- Pode-se fazer uma conversão de tipo explícita "cast" somente dentro de uma hierarquia de heranças.
- Usa-se "instandeof" para verificar uma hierarquia antes de fazer a conversão de tipo explícita de um objeto de superclasse para uma subclasse.

Classes Abstratas

Ao subir na hierarquia de heranças, as classes tornam-se mais genéricas e, provavelmente, mais abstratas. Em algum ponto, a classe ancestral torna-se tão geral que acaba sendo vista mais como um modelo pra outras classes do que uma classe com instâncias específicas que são usadas. Considere, por exemplo, um sistema de mensagens eletrônicas que integre e-mail, fax e correio de voz. Ele terá de ser capaz de lidar com mensagens de texto e correio de voz.

Seguido os princípios da POO, o programa vai precisar de classes como "TextMessage" e "VoiceMessage". Evidentemente, uma caixa postal terá de armazenar uma mistura deses tipos de mensagens, de modo que possa acessá-las através de referências à classe mão "Message" comum a todas as demais. A hierarquia das heranças é mostrada abaixo:

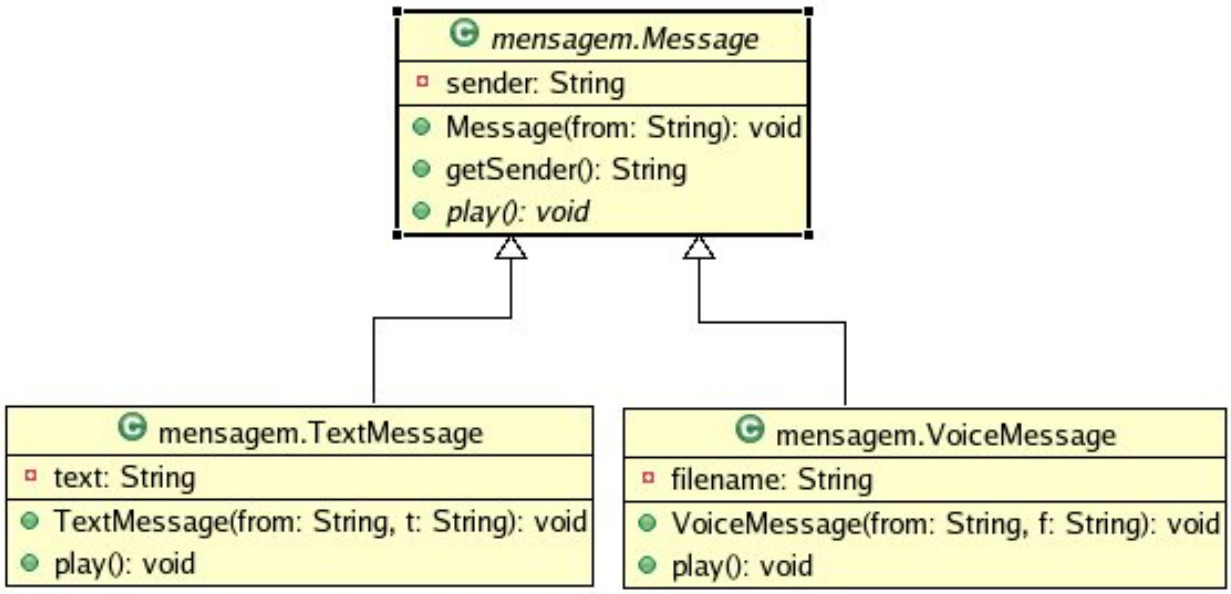


DIAGRAMA DE HERANÇA DAS CLASSES MENSAGENS

Porque se preocupar com um nível tão alto de abstração? A resposta é que isso torna o projeto das classes mais claro. No final das contas, uma das chaves da POO é entender como combinar e reunir operações comuns num nível mais alto na hierarquia de heranças. Em nosso caso, por exemplo, todas as mensagens têm um método comum denominado "play()". É fácil perceber como reproduzir uma mensagem de voz enviando-a ao alto falante, ou uma mensagem de texto exibindo-a em uma janela de texto, ou uma mensagem de fax imprimindo-a ou mostrando-a em uma janela gráfica, mas e como fazer para implementar o método "play()" na classe progenitora "Message"?

A resposta é que não se pode. Em Java, usa-se a palavra-chave "abstract" para indicar que um método não pode ser especificado nossa classe. Para aumentar a clareza, uma classe com um ou mais métodos abstratos precisa, ela mesma, ser declarada abstrata.

```
public abstract class Message {
	...
	public abstract void play();
}
```

Além dos métodos abstratos, as classes abstratas podem ter dados e métodos concretos. Por exemplo, a classe "Message" pode armazenar o remetente da mensagem postal e ter um método concreto que retorne o nome do remetente "sender".

```
public abstract class Message {
	private String sender;

	public Message(String from) {
		sender = from;
	}

	public String getSender() {
		return sender;
	}

	public abstract void play();
}
```

Um método abstrato promete que todos os descendentes não abstratos dessa classe abstrata irão implementar esse método abstrato. Os métodos abstratos funcionam como uma espécie de guardador de lugar para métodos que serão posteriormente implementados nas subclasses.

Uma classe pode ser declarada como abstrata mesmo sem ter métodos abstratos.

Não se podem criar objetos a partir de classes abstratas. Ou seja, se uma classe é declarada abstrata, então nenhum objeto dessa classe pode ser variado. Será necessário ampliar essa classe a fim de criar uma instância da classe. Observe ainda assim podem- se criar variáveis objeto de uma classe abstrata, embora essas variáveis tenham de fazer referência a um objeto de uma subclasse não abstrata.

```
Message msg = new VoiceMessage("bemvindo.wav");
```

Aqui, “msg” é uma variável do tipo abstrato “Message” que faz referência a uma instância da subclasse não abstrata “VoiceMessage”.

Para vermos uma realização concreta dessa classe abstrata e também do método “play”, vamos exemplificar o código para a classe “TextMessage”:

```
import javax.swing.JOptionPane;

public class TextMessage extends Message {
	private String text;

	public TextMessage(String from, String t) {
		super(from);
		text = t;
	}

	public void play() {
		JOptionPane.showMessageDialog(null, text, "M e n s a g e m : "
		+ getSender(), JOptionPane.INFORMATION_MESSAGE);
	}
}
```

Observe que nós precisamos fornecer somente uma definição concreta do método abstrato “play” na classe “TextMessage”.

Abaixo segue o código completo do programa exemplo de criação e leitura de mensagens.

Classe Message

```
public abstract class Message {
	private String sender;

	public Message(String from) {
		sender = from;
	}

	public String getSender() {
		return sender;
	}

	public abstract void play();
}
```

Classe TextMessage

```
import javax.swing.JOptionPane;

public class TextMessage extends Message {
	private String text;

	public TextMessage(String from, String t) {
		super(from);
		text = t;
	}

	public void play() {
		JOptionPane.showMessageDialog(null, text, "M e n s a g e m : "
			+ getSender(), JOptionPane.INFORMATION_MESSAGE);
	}
}
```

Classe VoiceMessage

```
import java.net.*;
import javax.sound.sampled.*;
import javax.swing.JOptionPane;

public class VoiceMessage extends Message {
	private String filename;

	public VoiceMessage(String from, String f) {
		super(from);
		filename = f;
	}

	@Override
	public void play() {
		try {
			URL u = new URL("file", "localhost", filename);

			AudioInputStream musica = AudioSystem.getAudioInputStream(u);

			Clip clip = AudioSystem.getClip();
			clip.open(musica);
			clip.setFramePosition(0);
			clip.start();
```

```
            } catch(Exception ex) {
                  JOptionPane.showMessageDialog(null,
                        "Não posso abrir o arquivo " + filename,
                        "E R R O", JOptionPane.ERROR_MESSAGE);
            }
      }
}
```

Classe Mailbox

```
public class Mailbox {
      private static final int MAX_MSG = 10;
      private int in = 0;
      private int out = 0;
      private int nmsg = 0;
      private Message[] messages = new Message[MAX_MSG];

      public Message next() {
            Message r = null;

            if(nmsg > 0) {
                  r = messages[out];
                  nmsg--;
                  out = (out + 1) % MAX_MSG;
            }
            return r;
      }

      public void insert(Message m) {
            if(nmsg < MAX_MSG) {
                  messages[in] = m
                  nmsg++;
                  in = (in + 1) % MAX_MSG;
            }
      }

      public String status() {
            String msg = "";

            if(nmsg == 0) msg = "Mailbox vazio";
            else if(nmsg ==1) msg = "1 mensagem";
            else if(nmsg < MAX_MSG) msg = nmsg + " mensagens";
```

```
            else msg = "Mailbox cheio”;

            return msg;
      }
}
```

Classe MailboxTest

```
import javax.swing.JOptionPane;

public class MailboxTest {
      public static void main(String[] args) {
            Mailbox mbox = new Mailbox();

            while(true) {
                  String from = “";
                  String msg = "";

                  int cmd = JOptionPane.showOptionDialog(null,
                              "Selecione a ação", "Mailbox “ +
                              mbox.status(), JOptionPane.DEFAULT_OPTION,
                              JOptionPane.QUESTION_MESSAGE, null,
                              new String[] {
                                    "Lê mensagem",
                                    "Cria mensagem Texto”,
                                    "Insere mensagem de Voz”,
                                    "Sair"
                              }, "Lê mensagem");

                  switch(cmd) {
                  case 0:
                        Message m = mbox.next();

                        if(m != null) {
                              JOptionPane.showMessageDialog(null,
                                    "Selecione OK para obter a mensagem”,
                                    "Mensagem de: " + m.getSender(),
                              JOptionPane.INFORMATION_MESSAGE);
                              m.play();
                        } else {
                              JOptionPane.showMessageDialog(null,
                                    "Não existem mensagens",
```

```
                                    "A t e n ç ã o”,
                                    JOptionPane.INFORMATION_MESSAGE);
                        }

                        break;
                case 1:
                        from = JOptionPane.showInputDialog(null,
                                "Informe seu nome", "Mensagem Texto”,
                                JOptionPane.QUESTION_MESSAGE);
                        msg = JOptionPane.showInputDialog(null,
                                "Entre com a mensagem", "Mensagem Texto”,
                                JOptionPane.QUESTION_MESSAGE);
                        mbox.insert(new TextMessage(from, msg));
                        break;
                case 2:
                        from = JOptionPane.showInputDialog(null,
                                "Informe seu nome", "Mensagem de Voz”,
                                JOptionPane.QUESTION_MESSAGE);
                        msg = JOptionPane.showInputDialog(null,
                                "Informe o nome do arquivo de Áudio”,
                                "Mensagem de Voz”,
                                JOptionPane.QUESTION_MESSAGE);
                        mbox.insert(new VoiceMessage(from, msg));
                        break;
                default:
                        System.exit(0);
                }
        }
    }
}
```

Acesso Protegido

Como sabemos, os campos de instância numa classe são melhor declarados como “private” e os métodos normalmente declarado como “public”. Quaisquer recursos declarados “private” não serão visíveis para as outras classes. Isto é verdade também para subclasses: uma subclasse não pode acessar os membros de dados privados de sua superclasse.

Há porém, casos em que se quer que uma subclasse tenha acesso a um método ou dado de uma classe progenitora. Nesse case, declara-se esse recurso como "protected" (protegido). Por exemplo, se a classe Empregado declarar o objeto "diaContratação" como "protected", em vez de "private", então os métodos de "Gerente" poderão acessá-lo diretamente.

Na prática, deve-se usar o atributo "protected com cautela. Suponha que sua classe seja usada por outros programadores e que você a tenha projetado com dados protegidos. Sem que você saiba, outros programadores poderão derivar classe a partir de sua classe e então começar a acessar seus campos de instância protegidos. Neste caso, você não poderá mais alterar a implementação de sua classe sem atrapalhar os outros programadores. Isso, evidentemente, é contrário ao espírito da POO, que enfatiza o encapsulamento dos dados.

Já os métodos protegidos fazem mais sentido. Uma classe pode declarar um método como "protected" se sua utilização for específica. Isso indica que as subclasses (as quais, presumivelmente, conhecem bem seus ancestrais) podem ser confiáveis na utilização do método corretamente, mas outras classes não.

Eis um resumo dos quatro modificadores de acesso em Java que controlam a visibilidade:

1. Visível somente para a classe – private
2. Visível para o mundo public
3. Visível para o pacote e todas as subclasses – protected
4. Visível para o pacote - "sem declaração de encapsulamento"

Interfaces

Uso de uma Superclasse Abstrata

Suponha que queiramos escrever uma rotina de ordenamento genérica que trabalhe com vários tipos diferentes de objetos Java. Agora você já sabe como organizar isso em um modelo orientado a objetos. Você começa com uma classe abstrata "Sortable" contendo um método "compareTo" que determina se ou não um objeto ordenável é menor que, igual ou maior que outro.

Primeiro, deve-se implementar um algoritmo de ordenamento genérico. Eis a seguir uma implementação de uma variante do algoritmo "Shell", para ordenar um array de objetos "Sortable". Não se preocupe em saber exatamente como esse algoritmo funciona. Nós simplesmente o escolhemos porque é simples de implementar. Observe sim que o algoritmo analisa cada elemento do array e os compara usando o método "compareTo", ordenando-os em seguida.

```
public abstract class Sortable {
	public abstract int compareTo(Sortable b);
}

public class ArrayAlg {
	public static void shellSort(Sortable[] a) {
		int n = a.length;
		int incr = n / 2;

		while(incr >= 1) {
			for(int i = incr;i < n;i++) {
				Sortable temp = a[i];
				int j = i;

				while(j >= incr &&
					temp.compareTo(a[j - incr]) < 0) {
					a[j] = a[j - incr];
					j -= incr;
				}

				a[j] = temp;
			}
			incr /= 2;
		}
	}
}
```

Essa rotina de ordenamento pode ser usada em todas as subclasses da classe abstrata Sortable (sobrepondo o método "compareTo" nas subclasses).

Por exemplo, para ordenar um array de empregados (ordenando-os por qualquer coisa, como seus salários por exemplo), é necessário:

1. Derivar "Empregado" a partir de "Sortable".
2. Implementar o método "compareTo" para os empregados.
3. Chamar "ArrayAlg.shellSort" no array de empregados.

Eis a seguir um exemplo do código extra necessário para fazer isso em nossa classe "Empregado":

```
public class Empregado extends Sortable {
	...
```

```
	@Override
	public int compareTo(Sortable b) {
		int resultado = 0;
		Empregado ep = (Empregado)b;

		if(salario < ep.salario) resultado = -1;
		else if(salario > ep.salario) resultado = 1;

		return resultado;
	}
}
```

Abaixo segue o código completo do algoritmo de ordenamento genérico e a classificação do array de empregados.

Classe EmpregadoSortTest

```
import java.util.*;

public class EmpregadoSortTest {
	public static void main(String[] args) {
		Empregado[] grupo = new Empregado[3];

		Calendar dt = new GregorianCalendar(1989, 10, 1);
		grupo[0] = new Empregado("Chico da Silva", 1500, dt.getTime());

		dt = new GregorianCalendar(1987, 12, 15);
		grupo[1] = new Gerente("José de Oliveira", 2500, dt.getTime());

		dt = new GregorianCalendar(1990, 3, 18);
		grupo[2] = new Empregado("Ana Ribeiro", 1700, dt.getTime());

		ArrayAlg.sort(grupo);

		for(int i = 0;i < grupo.length;i++) {
			System.out.println(grupo[i]);
		}
	}
}
```

Classe ArrayAlg

```
public class ArrayAlg {
	public static void shellSort(Sortable[] a) {
		int n = a.length;
		int incr = n / 2;

		while(incr >= 1) {
			for(int i = incr;i < n;i++) {
				Sortable temp = a[i]; int j = i;

				while(j >= incr && temp.compareTo(a[j - incr]) < 0) {
					a[j] = a[j - incr];
					j -= incr;
				}

				a[j] = temp;
			}

			incr /= 2;
		}
	}
}
```

Classe Sortable

```
public abstract class Sortable {
	public abstract int compareTo(Sortable b);
}
```

Classe Empregado (Modificada)

```
import java.util.*;

public class Empregado extends Sortable {
	private String nome;
	private double salario;
	private Date dataContratacao;

	public Empregado(String nome, double sal, Date data) {
		this.nome = nome;
		this.salario = sal;
		this.dataContratacao = data;
	}
	public void aumentaSalario(double porPercentual) {
```

```
            salario *= 1 + porPercentual / 100;
      }

      public int anoContratacao() {
            Calendar dt = Calendar.getInstance();
            dt.setTime(dataContratacao);
            return dt.get(Calendar.YEAR);
      }

      public String toString() {
            return String.format("%s %,.2f %d", nome, salario,
                  anoContratacao());
      }

      @Override
      public int compareTo(Sortable b) {
            int resultado = 0;
            Empregado ep = (Empregado)b;

            if(salario < ep.salario) resultado = -1;
            else if(salario > ep.salario) resultado = 1;

            return resultado;
      }
}
```

Class Gerente

```
import java.util.*;

public class Gerente extends Empregado {
      private String nomeSecretaria;

      public Gerente(String nome, double sal, Date data) {
            super(nome, sal, data);
            nomeSecretaria = "";
      }

      public String getNomeSecretaria() {
            return nomeSecretaria;
      }
      public void setNomeSecretaria(String nome) {
```

```
            nomeSecretaria = nome;
      }
}
```

Uso de Interfaces

Infelizmente, há um problema importante ao se usar uma classe abstrata para expressar uma propriedade genérica. Eis a seguir um exemplo onde esse problema surge: considere uma classe "Tile" que modela janelas lado a lado (tiled) sobre a área de trabalho (tela) do computador. As janelas dado a lado são na verdade retângulos com uma ordem z (z-order). Ora, as janelas com ordem z maior são exibidas na frente daquelas com ordem z menor. Para reutilizar código, nós derivamos "Tile" de "Rectangle", uma classe que já está definida no pacote "java.awt".

```
import java.awt.Rectangle;

public class Tile extends Rectangle {
      private int z;

      public Tile(int x, int y, int width, int height, int z) {
            super(x, y, width, height);
            this.z = z;
      }
}
```

Agora nós gostaríamos de ordenar um array de janelas lado a lado comparando suas ordens z. Se tentarmos aplicar o procedimento para tornar essas janelas ordenáveis, ficaríamos presos no primeiro passo. Nós não podemos derivar "Tile" de "Sortable", pois esta já deriva de "Rectangle"!

O problema é que, em Java, uma classe só pode ter uma superclasse. Outras linguagens de programação, em particular C++, permitem que uma classe tenha mais de uma superclasse. Esse recurso é chamado de herança múltipla.

Em vez disso, a linguagem Java introduz a noção de interfaces para recuperar grande parte da funcionalidade que a herança múltipla oferece. Os projetistas Java optaram por esse caminho porque as heranças múltiplas tornam os compiladores muito complexos (como em C++) ou muito ineficientes (como em Eiffel). As interfaces também são o método preferido de se implementar funções de "callback" (chamada de retorno) em Java.

Bom, e o que é uma interface afinal? Essencialmente, é uma promessa de que uma classe vai implementar certos métodos com certas características. Usa-se inclusive a palavra-chave "implements" para indicar que a classe vai manter essa promessa. A maneira com que esses métodos são implementados depende da classe, evidentemente. O que é importante, até onde interessa ao compilador, é que os métodos tenham as características corretas.

Por exemplo, a biblioteca padrão define uma interface chamada "Comparable" que poderia ser usada por qualquer classe cujos elementos possam ser comparados. O código da interface "Comparable" se parece com este:

```
public interface Comparable {
	public int comparaTo(Object b);
}
```

Esse código promete que qualquer classe que implemente a interface "Comparable" terá um método "compareTo". Evidentemente, a maneira pela qual o método "compareTo" funciona (ou mesmo se funciona ou não conforme esperado) em uma classe específica vai depender da classe que está implementando a interface "Comparable". O ponto a ser observado aqui é que qualquer classe pode prometer implementar "Comparable" - não importa se sua superclasse promete ou não o mesmo. Todos os descendentes de uma classe dessa haveriam de implementar "Comparable" automaticamente, pois todos eles teriam acesso a um método 'compareTo" com as características corretas.

Para informar à linguagem Java que uma classe implementa a interface "Comparable", ela deve ser definida da seguinte forma:

```
public class Tile extends Rectangle implements Comparable
```

Depois, basta implementar um método "compareTo" dentro da classe:

```
public class Tile extends Rectangle implements Comparable<Tile> {
	...

	public int compareTo(Tile o) {
		return z - o.z;
	}
}
```

Convenientemente, a classe "Arrays" do Java fornece um algoritmo de ordenamento para um array de objetos "Comparable". Nós podemos usar isso para ordenar um array de janelas lado a lado:

```
Tile[] a = new Tile[20];
      ...
Arrays.sort(a);
```

Abaixo segue o fonte do exemplo de janelas lado a lado.

Classe Tile

```
import java.awt.Rectangle;

      public class Tile extends Rectangle implements Comparable<Tile> {
            private int z;

            public Tile(int x, int y, int width, int height, int z) {
                  super(x, y, width, height);
                  this.z = z;
      }

      @Override
      public String toString() {
            return super.toString() + "z=" + z + “]";
      }

      public int compareTo(Tile o) {
            return z - o.z;
      }
}
```

Classe TileTest

```
import java.util.*;

public class TileTest {
      public static void main(String[] args) {
            Tile[] a = new Tile[20];

            for(int i = 0;i < a.length;i++)
                  a[i] = new Tile(i, i, 10, 20, (int)(100 * Math.random()));

            Arrays.sort(a);
            for(int i = 0;i < a.length;i++) System.out.println(a[i]);
      }
}
```

Interfaces em Evolução

Um problema no desenvolvimento do Java 8 foi como evoluir interfaces. O Java 8 incluiu lambdas e vários outros recursos na linguagem Java que tornaram desejável adaptar algumas das interfaces existentes na biblioteca Java. Mas como você evolui uma interface sem quebrar todo o código existente que usa essa interface?

Imagine que você tenha uma interface VarinhaMagica na sua biblioteca existente:

```
public interface VarinhaMagica {
      void executeMagica();
}
```

Essa interface já foi usada e implementada por muitas classes em muitos projetos. Mas agora você tem novas funcionalidades realmente boas e gostaria de adicionar um novo método realmente útil:

```
public interface VarinhaMagica {
      void executeMagica();
      void executeMagicaAvancada();
}
```

Se você fizer isso, todas as classes que implementaram anteriormente essa interface quebram, porque elas precisam fornecer uma implementação para esse novo método. Assim, à primeira vista, parece que você está preso: ou você quebra o código de usuário existente (o que você não quer fazer) ou está fadado a ficar com suas bibliotecas antigas sem a chance de melhorá-las facilmente. (Na verdade, existem algumas outras abordagens que você poderia tentar, como estender interfaces em subinterfaces, mas elas têm seus próprios problemas, que eu não discuto aqui.) O Java 8 criou um truque inteligente para obter o melhor de ambos mundos: a capacidade de adicionar interfaces existentes sem quebrar o código existente. Isso é feito usando métodos padrão e métodos estáticos, que discuto agora.

Métodos Padrão

Os métodos padrão são métodos em interfaces que possuem um corpo de método - a implementação padrão. Eles são definidos usando o modificador padrão no início da assinatura do método e eles têm um corpo de método completo:

```
public interface VarinhaMagica {
      void executeMagica();
      default void executeMagicaAvancada() {
      ... // algum código aqui
      }
}
```

As classes que implementam essa interface agora têm a chance de fornecer sua própria implementação para esse método (substituindo-a) ou podem ignorar completamente esse método, caso em que recebem a implementação padrão da interface. O código antigo continua a funcionar, enquanto o novo código pode usar essa nova funcionalidade.

Métodos estáticos

As interfaces agora também podem conter métodos estáticos com implementações. Estes são definidos usando o modificador estático usual no início da assinatura do método. Como sempre, ao escrever interfaces, o modificador público pode ficar de fora, porque todos os métodos e todas as constantes nas interfaces são sempre públicos.

Classes abstratas x Interfaces

Como você pode ver, as classes e interfaces abstratas tornaram-se bastante semelhantes agora. Ambos podem conter métodos e métodos abstratos com implementações, embora a sintaxe seja diferente. Ainda existem algumas diferenças (por exemplo, classes abstratas podem ter campos de instância, enquanto interfaces não podem), mas essas diferenças suportam um ponto central: desde o lançamento do Java 8, você tem herança múltipla (via interfaces) que pode conter código!

Como sempre, os projetistas de Java criaram as seguintes regras práticas e sensatas para lidar com esses problemas:

- Herdar vários métodos abstratos com o mesmo nome não é um problema - eles são vistos como o mesmo método.
- A herança de campos de diamante - um dos problemas difíceis - é evitada, porque as interfaces não podem conter campos que não sejam constantes.
- Herdar métodos e constantes estáticos (que também são estáticos por definição) não é um problema, porque eles são prefixados pelo nome da interface quando são usados, portanto seus nomes não entram em conflito.
- Herdar de diferentes interfaces vários métodos padrão com a mesma assinatura e implementações diferentes é um problema. Mas aqui o Java escolhe uma solução muito mais pragmática do que algumas outras linguagens: em vez de definir uma nova construção de linguagem para lidar com isso, o compilador apenas relata um erro. Em outras palavras, é problema seu. Java apenas diz: "Não faça isso".

Interfaces são um recurso poderoso em Java. Eles são úteis em muitas situações, inclusive para definir contratos entre diferentes partes do programa, definindo tipos para o despacho dinâmico, separando a definição de um tipo de sua implementação e permitindo a herança múltipla em Java. Eles são muito úteis em seu código; você deve se certificar de que entende bem o comportamento deles.

Os novos recursos de interface no Java 8, como os métodos padrão, são mais úteis quando você escreve bibliotecas; eles são menos propensos a serem usados no código do aplicativo. No entanto, as bibliotecas Java agora fazem uso extensivo delas, portanto, saiba o que elas fazem. O uso cuidadoso das interfaces pode melhorar significativamente a qualidade do seu código.

Módulos

A modularidade adiciona um nível mais alto de agregação acima dos pacotes. O novo elemento-chave da linguagem é o módulo - um grupo reutilizável e único de pacotes relacionados, bem como recursos (como imagens e arquivos XML) e um descritor de módulo que especifica

- o *nome* do módulo
- as *dependências* do módulo (isto é, outros módulos em que este módulo depende)
- os pacotes explicitamente disponibilizados para outros módulos (todos os outros pacotes no módulo estão *implicitamente indisponíveis* para outros módulos)
- os serviços que oferece
- os serviços que consome
- para que outros módulos permite *reflexão*

Introdução

A plataforma Java SE existe desde 1995. Atualmente, existem aproximadamente 10 milhões de desenvolvedores usando tudo isso, desde pequenos aplicativos para dispositivos com recursos limitados, como os da Internet das Coisas (IoT) e de outros dispositivos incorporados, até grandes sistemas críticos de negócios e de missão crítica. Existem enormes quantidades de código legado por aí, mas até agora, a plataforma Java tem sido basicamente uma solução monolítica de tamanho único para todos. Ao longo dos anos, tem havido vários esforços voltados para a modularização do Java, mas nenhum é amplamente utilizado - e nenhum deles poderia ser usado para modularizar a plataforma Java.

A modularização da plataforma Java SE tem sido um desafio para implementar, e o esforço levou muitos anos. JSR 277: O Java Module System foi originalmente proposto em 2005 para o Java 7. Este JSR foi posteriormente substituído pelo JSR 376: Java Platform Module System e direcionado para o Java 8. A plataforma Java SE agora é modularizada no Java 9, mas somente após o Java 9 foi adiada até setembro de 2017.

Objetivos

De acordo com a JSR 376, os principais objetivos da modularização da plataforma Java SE são:

- Configuração confiável - A modularidade fornece mecanismos para declarar explicitamente as dependências entre os módulos de uma maneira que seja reconhecida tanto no tempo de compilação quanto no tempo de execução. O sistema pode percorrer essas dependências para determinar o subconjunto de todos os módulos necessários para dar suporte ao seu aplicativo.
- Encapsulamento forte - Os pacotes em um módulo só podem ser acessados por outros módulos se o módulo exportá-los explicitamente. Mesmo assim, outro módulo não pode usar esses pacotes a menos que explicitamente afirme que requer os recursos do outro módulo. Isso melhora a segurança da plataforma porque menos classes são acessíveis a invasores em potencial. Você pode descobrir que, considerando a modularidade, ajuda a criar designs mais limpos e mais lógicos.
- Plataforma Java escalonável - Anteriormente, a plataforma Java era um monólito que consistia em um grande número de pacotes, tornando difícil desenvolver, manter e evoluir. Não poderia ser facilmente subdividido. A plataforma agora é modularizada em 95 módulos (esse número pode mudar conforme o Java evolui). Você pode criar execuções personalizadas que consistem apenas em módulos necessários para seus aplicativos ou para os dispositivos que você está segmentando. Por exemplo, se um dispositivo não suportar GUIs, você poderá criar um tempo de execução que não inclua os módulos da GUI, reduzindo significativamente o tamanho do tempo de execução.
- Maior integridade da plataforma - Antes do Java 9, era possível usar muitas classes na plataforma que não eram destinadas às classes de um aplicativo. Com um forte encapsulamento, essas APIs internas são realmente encapsuladas e ocultas dos aplicativos que usam a plataforma. Isso pode tornar problemático migrar o código legado para o Java 9 modularizado se o seu código depender de APIs internas.
- Desempenho aprimorado - A JVM usa várias técnicas de otimização para melhorar o desempenho do aplicativo. O JSR 376 indica que essas técnicas são mais eficazes quando se sabe de antemão que os tipos necessários estão localizados apenas em módulos específicos.

Listando os Módulos do JDK

JEP 200: O JDK MODULAR

JEP 201: CÓDIGO FONTE MODULAR

JEP 220: IMAGENS DE TEMPO MODULAR

JEP 260: ENCAPSULAR APIS MAIS INTERNO

JEP 261: SISTEMA DE MÓDULO

JEP 275: EMBALAGEM DE APLICAÇÃO DE JAVA MODULAR

JEP 282: JLINK: O LINKER DE JAVA

JSR 376: SISTEMA DE MÓDULO DE PLATAFORMA JAVA

JSR 379: JAVA SE 9

JEPS E JSRS DE MODULARIDADE JAVA

Um aspecto crucial do Java 9 é dividir o JDK em módulos para suportar várias configurações. (Consulte "JEP 200: O JDK Modular". Todos os JEPs e JSRs de modularidade Java são mostrados na Tabela 1.) Usando o comando java da pasta bin do JDK com a opção --list-modules, como em:

```
java --list-modules
```

lista o conjunto de módulos do JDK, que inclui os módulos padrão que implementam o Java Language SE Specification (nomes iniciados por java), módulos JavaFX (nomes iniciados por javafx), módulos específicos do JDK (nomes iniciados por jdk) e módulos específicos do Oracle com oracle). Cada nome de módulo é seguido por uma sequência de versão - @9 indica que o módulo pertence ao Java 9.

Declarações do Módulo

Como mencionamos, um módulo deve fornecer um descritor de módulo - metadados que especificam as dependências do módulo, os pacotes que o módulo disponibiliza para outros módulos e muito mais. Um descritor de módulo é a versão compilada de uma declaração de módulo que é definida em um arquivo chamado module- info.java. Cada declaração de módulo começa com a palavra-chave module, seguida por um nome de módulo exclusivo e um corpo de módulo entre chaves, como em:

```
module nome_do_modulo {
}
```

O corpo da declaração do módulo pode estar vazia ou pode conter várias directivas do módulo, incluindo `requires`, `exports`, `provides` ... `with`, `uses` e `opens` (cada um dos quais nós discutiremos). Como você verá mais adiante, a compilação da declaração do módulo cria o descritor do módulo, que é armazenado em um arquivo chamado module-info.class na pasta raiz do módulo. Aqui nós introduzimos brevemente cada diretiva de módulo. Depois disso, apresentaremos as declarações reais do módulo.

As palavras chave `exports`, `module`, `open`, `opens`, `provides`, `requires`, `uses`, `with`, bem como `to` e `transitive`, o que nós introduziremos mais tarde, são palavras- chave restritas. Eles são palavras-chave apenas em declarações de módulo e podem ser usados como identificadores em qualquer outro lugar em seu código.

requires. Uma diretiva de módulo `requires` especifica que esse módulo depende de outro módulo - esse relacionamento é chamado de dependência de módulo . Cada módulo deve declarar explicitamente suas dependências. Quando o módulo A `requires` módulo B, o módulo A é dito para ler o módulo B e o módulo B é lido pelo módulo A. Para especificar uma dependência em outro módulo, use `requires`, como em:

```
requires nome_do_modulo;
```

Há também uma diretiva `requires static` para indicar que um módulo é necessário em tempo de compilação, mas é opcional no tempo de execução. Isso é conhecido como uma dependência opcional e não será discutido nesta introdução.

requires transitive - implied readability . Para especificar uma dependência em outro módulo e para garantir que outros módulos que leiam seu módulo também leiam essa dependência - conhecida como legibilidade implícita - use `requires transitive`, como em:

```
requires transitive nome_do_modulo;
```

Considere a seguinte diretiva da declaração do módulo `java.desktop`:

```
requires transitive java.xml;
```

Neste caso, qualquer módulo que leia `java.desktop` também lê implicitamente `java.xml`. Por exemplo, se um método do módulo `java.desktop` retornar um tipo do módulo `java.xml`, o código nos módulos que ler `java.desktop` se tornará dependente `java.xml`. Sem a declaração do módulo da diretiva na `java.desktop requires transitive`, tais módulos dependentes não compilarão a menos que eles leiam explicitamente `java.xml` .

De acordo com a JSR 379 , os módulos padrão do Java SE devem conceder legibilidade implícita em todos os casos, como o descrito aqui. Além disso, embora um módulo padrão Java SE possa depender de módulos não padrão, ele não deve conceder legibilidade implícita a eles. Isso garante que o código que depende apenas dos módulos padrão do Java SE seja portátil nas implementações do Java SE.

exports e exports ... to. Uma diretiva de módulo exports especifica um dos pacotes do módulo cujos tipos `public` (e seus tipos aninhados `public` e `protected`) devem estar acessíveis ao código em todos os outros módulos. Uma diretiva `exports...to` permite que você especifique em uma lista separada por vírgula com precisão qual código de módulo ou módulo pode acessar o pacote exportado - isso é conhecido como exportação qualificada.

uses. Uma diretiva de módulo `uses` especifica um serviço usado por este módulo - tornando o módulo um consumidor de serviço. Um serviço é um objeto de uma classe que implementa a interface ou estende a classe abstrata especificada na diretiva `uses`.

provides... with. Uma diretiva de módulo `provides...with` especifica que um módulo fornece uma implementação de serviço - tornando o módulo um provedor de serviços . A parte `provides` da diretiva especifica uma interface ou classe abstrata listada na diretiva `uses` de um módulo e a parte `with` da diretiva especifica o nome da classe do provedor de serviços que implementa a interface ou estende uma classe abstrata.

open, **open** e **opens ... to**. Antes do Java 9, a reflexão poderia ser usada para aprender sobre todos os tipos em um pacote e todos os membros de um tipo - até mesmo seus membros private - independente se você queria ou não permitir esse recurso. Assim, nada foi verdadeiramente encapsulado.

A motivação chave do sistema de módulos é um forte encapsulamento. Por padrão, um tipo em um módulo não é acessível a outros módulos, a menos que seja um tipo público e você exporte seu pacote. Você expõe apenas os pacotes que deseja expor. Com o Java 9, isso também se aplica à *reflection*.

Permitindo acesso somente em tempo de execução a um pacote. Uma diretiva de módulo de abertura no formato:

```
opens pacote
```

indica que um determinado pacote de tipos public (e seus tipos aninhados public e protected) são acessíveis ao código em outros módulos em apenas tempo de execução. Além disso, todos os tipos no pacote especificado (e todos os membros dos tipos) são acessíveis via *reflection*.

Permitir acesso em tempo de execução apenas a um pacote por módulos específicos. Uma diretiva de módulo `opens...to` no formato:

```
opens pacote to lista_separada_por_virgula_de_modulos
```

indica que um determinado pacote de tipos public (e seus tipos aninhados public e protected) são acessíveis ao código nos módulos listados em apenas tempo de execução. Todos os tipos no pacote especificado (e todos os membros dos tipos) são acessíveis por meio de reflexão para codificar nos módulos especificados.

Permitir acesso somente em tempo de execução a todos os pacotes em um módulo. Se todos os pacotes em um determinado módulo devem estar acessíveis em tempo de execução e através de reflection para todos os outros módulos, você pode abrir (`open`) o módulo inteiro, como em:

```
open module nome_do_modulo {
      // diretivas do modulo
}
```

Funções Matemáticas

A linguagem Java possui uma classe com diversos métodos especializados em realizar cálculos matemáticos. Para realizar esses cálculos, são utilizados os métodos da classe Math.

A classe Math define duas constantes matemáticas, sendo Math.PI o valor do pi (3.14159265358979323846) e Math.E que se refere ao valor da base para logaritmos naturais (2.7182818284590452354).

Os métodos mais comuns da classe Math são:

Math.ceil()

Este método tem como função realizar o arredondamento de um número do tipo double para o seu próximo inteiro.

```
public class ExemploCell {
	public static void main(String[] args) {
		double a = 5.2, b = 5.6, c = -5.8;

		System.out.println(Math.cell(a)); // apresentará 6.0
		System.out.println(Math.cell(b)); // apresentará 6.0
		System.out.println(Math.cell(c)); // apresentará -5.0
	}
}
```

Math.floor()

Assim com ceil, o método floor também é utilizado para arredondar um determinado número, mas para seu inteiro anterior.

```
public class ExemploFloor {
	public static void main(String[] args) {
		double a = 5.2, b = 5.6, c = -5.8;

		System.out.println(Math.floor(a)); // apresentará 5.0
		System.out.println(Math.floor(b)); // apresentará 5.0
		System.out.println(Math.floor(c)); // apresentará -6.0
	}
}
```

Math.max()

Utilizado para verificar o maior valor entre dois números, que podem ser do tipo double, float, int ou long.

```
public class ExemploMax {
	public static void main(String[] args) {
		int a = 10; b = 15;
		double c = -5.9, d = -4.5;

		System.out.println(Math.max(a, b)); // apresentará 15
		System.out.println(Math.max(c, d)); // apresentará -4.5
		System.out.println(Math.max(a, c)); // apresentará 10.0
	}
}
```

Math.min()

O método min fornece o resultado contrário do método max, sendo então utilizado para obter o valor mínimo entre dois números.

```
public class ExemploMin {
	public static void main(String[] args) {
		int a = 10; b = 15;
		double c = -5.9, d = -4.5;

		System.out.println(Math.min(a, b)); // apresentará 10
		System.out.println(Math.min(c, d)); // apresentará -5.9
		System.out.println(Math.min(a, c)); // apresentará -5.9
	}
}
```

Math.sqrt()

Quando há necessidade de calcular a raiz quadrada de um determinado número, utiliza- se o método sqrt, porém o número do qual se deseja extrair a raiz quadrada deve ser do tipo double e o resultado obtido também será um número do tipo double.

```
public class ExemploSqrt {
	public static void main(String[] args) {
		double a = 900, b = 30.25;

		System.out.println(Math.sqrt(a)); // apresentará 30.0
		System.out.println(Math.sqrt(b)); // apresentará 5.5
	}
}
```

Math.pow()

Este método eleva um determinado número ao quadrado ou a qualquer outro valor de potência. Os argumentos e o resultado são do tipo double.

```
public class ExemploPow {
	public static void main(String[] args) {
		double a = 5.5, b = 2;

		System.out.println(Math.pow(a, b));	// apresentará 30.25
		System.out.println(Math.pow(25, 0.5));	// apresentará 5.0
		System.out.println(Math.pow(1234, 0));	// apresentará 1.0
	}
}
```

Math.random()

O método random retorna um número do tipo double com valor entre 0 e 1. Este resultado é gerado randomicamente com uma distribuição aproximadamente uniforme entre 0 e 1.

Para obtermos números aleatórios uma faixa de valores entre 0 e 99, por exemplo, necessitamos multiplicar o resultado de Math.random() por 100.

```
public class ExemploRandom {
	public static void main(String[] args) {
		for(int i = 0;i < 3;i+) {
			int num = (int)(Math.random() * 100);
			System.out.println(num);
		}
	}
}
```

Manipulação de Strings

Uma string é um tipo texto que corresponde à união de um conjunto de caracteres. Em Java toda a sequência de caracteres é representada como uma instância da classe String, é constante e o texto contido em um objeto desta classe não pode ser modificado depois de sua criação.

Seus construtores são:

```
char[] arr = { ‘a’, ‘b’, ‘c’, ‘d’ };

String a = new String();
String b = new String(arr);
String c = new String(“Java”);
```

Por ser um objeto imutável, as Strings podem ser compartilhadas, assim, a instanciação de um objeto desta classe pode ser feita sem a invocação de seu construtor.

```
String d = “Texto”);
```

A API java também oferece suporte especial para a conversão de quaisquer tipos de objetos para um objeto da classe String , sendo possível obter a representação textual de objetos de quaisquer classes através de seu método toString() , que é definido na classe Object.

```
Double db = new Double(5.5);
String st = db.toString();
```

A classe String também implementa diversas versões de um método static, chamado valueOf(), através do qual é possível obter uma representação textual de um dado de qualquer um dos tipos primitivos.

```
int it = 5;
String st = String.valueOf(it);
```

Além do exposto acima String tem outros métodos:

length()

O método length é utilizado para retornar o tamanho de uma determinada string, incluindo também os espaços em branco contidos nela.

```
String st = “Texto Explicativo”;
System.out.println(st.length()); // apresentará 17
```

charAt()

Utilizado para se obter um caractere de determinada string de acordo com o índice informado. Podemos imaginar a String como um array de caracteres cuja primeira posição é 0.

```
String st = “Texto Explicativo”;
System.out.println(st.charAt()); // apresentará p
```

toUpperCase() e toLowerCase()

São utilizados para transformar todas os caracteres de uma determinada string em maiúsculas ou minúsculas respectivamente.

```
String st = “Texto Explicativo”;
System.out.println(st.toUpperCase()); // apresentará  TEXTO EXPLICATIVO
System.out.println(st.toLowerCase()); // apresentará  texto explicativo
```

substring()

Retorna uma cópia de caracteres de uma string a partir de dois índices interos. O primeiro argumento indica a partir de que posição se inicia a cópia, o segundo é opcional e especifica a posição em que termina a cópia dos caracteres.

```
String st = “Texto Explicativo”;
System.out.println(st.substring(2));      // apresentará  xto Explicativo
System.out.println(st.substring(0, 7));   // apresentará  Texto E
System.out.println(st.substring(3, 7));   // apresentará  to E
```

Operações com Datas

Operações com datas são frequentes em um grande número de sistemas. São raros, por exemplo, os aplicativos de banco de dados que não necessitam manipular uma única data sequer. Geralmente, esse tipo de aplicativo precisa lidar com datas em diversas de suas funções.

Para compreender como tratar de datas em um programa qualquer, é preciso identificar as operações fundamentais que precisam ser realizadas com elas: a formatação e a conversão. A formatação diz respeito à transformação de uma data em um texto, definindo-se a sua forma de exibição. Para estas operações normalmente são utilizadas um conjunto de classes: Date, Calendar, DateFormat, SimpleDateFormat e Formatter.

Classe Date

A classe Date representa um instante específico no tempo, com precisão de milissegundos, e pode ser utilizada para representar datas. Quando fora criada, essa classe continha duas funções adicionais: permitia a interpretação de datas em termos de ano, mês e dia e também realizava operações de formatação e conversão. Mas os seus métodos não eram adequados à internacionalização e, desde o JDK 1.1 eles foram depreciados. A partir desta data, essa classe passou a ser usada como repositório da data. Agora, deve-se utilizar as classes DateFormat e SimpleDateFormat para realizar operações de formatação e conversão de datas e as classes Calendar e GregorianCalendar para interpretar uma data em termos de suas partes componentes.

Podemos obter a instância de um objeto do tipo Date da seguinte forma:

```
Date agora = new Date();
```

Este construtor cria um objeto Date com a representação da data e hora no momento da construção deste. Internamente um objeto Date armazena a data como um número do tipo long, cujo valor representa a quantidade de milissegundos entre a data atual e a sua data de referência (1/1/1970 00:00h).

Também podemos construir um objeto do tipo Date a partir de uma representação de data no formato long, conforme a descrição acima.

```
Date hoje = new Date(1082345234000L); // 19/04/2004 00:27:14H
```

Embora a construção acima seja possível, provavelmente é melhor utilizarmos outra alternativa para a construção de datas:

```
Calendar c = new GregorianCalendar(2004,Calendar.APRIL,19,00,27,14);
Date hoje = c.getTime();
```

Esta construção tem o mesmo efeito da construção anterior, e é mais fácil seu entendimento.

Os métodos da classe Date são:

after()

Este método recebe um objeto do tipo Date como argumento e retorna um valor booleano indicando se a data representada pelo objeto atual é posterior a data do objeto passado pelo argumento.

```
public class ExemploAfter {
	public static void main(String[] args) {
		Date agora = new Date();

		Calendar c = new GregorianCalendar(2004, Calendar.APRIL,
					19, 00, 27, 14);
		Date data = c.getTime();

		if(agora.after(date)) {
			System.out.println("A data: " + agora +
				" é posterior a data: " + data);
		}
	}
}
```

before()

O método before recebe um objeto do tipo Date como argumento e retorna um valor booleano indicando se a data representada pelo objeto atual é anterior a data do objeto passado pelo argumento.

```
public class ExemploBefore {
	public static void main(String[] args) {
		Date agora = new Date();

		Calendar c = new GregorianCalendar(2004, Calendar.APRIL,
					19, 00, 27, 14);
		Date data = c.getTime();
```

```
            if(agora.before(date)) {
                  System.out.println("A data: " + agora +
                        " é anterior a data: " + data);
            }
      }
}
```

getTime()

Este método retorna um número do tipo long, cujo valor representa a quantidade de milissegundos entre a data atual e a sua data de referência (1/1/1970 00:00h).

Com este valor podemos calcular a diferença entre datas, pois, se subtrairmos o valor em milissegundos de duas datas, e dividirmos por 1000 milissegundos / 60 segundos / 60 minutos / 24 horas, obteremos o número de dias que representa esta diferença.

```
long agoraMilisec = agora.getTime();
long dataMilicec = data.getTime();
long dias = (agoraMilisec - dataMilisec) / 1000 / 60 / 60 / 24;
```

Classe Calendar e GregorianCalendar

As classes Calendar e GregorianCalendar oferecem mecanismos adequados para realização de cálculos com datas ou para identificação das propriedades de uma data, como, por exemplo, para identificar o dia da semana, o dia do mês em relação ao ano etc.

Para isso estas classes convertem um tipo Date armazenando internamente em um série de campos. Elas também possuem métodos para recuperar (get) e armazenar (set) os valores correspondentes a datas e horas, por meio de um argumento fornecido que identifica o campo a ser manipulado. Por exemplo, para recuperar o dia do ano, pode ser usada a sintaxe get(Calendar.YEAR). Neste caso, a sintaxe define que o campo a ser manipulado é o ano (YEAR) da data.

Os principais campos usados são mostrados na tabela a seguir.

Campo	Descrição
DAY_OF_MONTH	Dia do mês (1 a 31)
DAY_OF_WEEK	Dia da semana (0 = domingo, 6 = sábado)
DAY_OF_WEEK_IN_MONTH	Semana do mês (1 a 5) corrente. Diferente em relação a WEEK_OF_MONTH porque considere apenas a semana cheia
DAY_OF_YEAR	Dias decorridos no ano corrente
HOUR	Hora do dia (manhã ou tarde) (0 a 11)

Campo	Descrição
HOUR_OF_DAY	Hora do dia (0 a 23)
MILLISECOND	Milissegundos em relação ao segundo corrente
MINUTE	Minutos em relação à hora corrente
MONTH	Mês em relação ao ano corrente
SECOND	Segundos em relação ao minuto corrente
WEEK_OF_MONTH	Semana em relação ao mês corrente (1 a 5)
WEEK_OF_YEAR	Semana em relação ao ano corrente
JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER	Mês correspondente ao ano
MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY	Dia correspondente à semana

Para obtermos uma instância de Calendar necessitamos chamar o método getInstance() de Calendar ou utilizar um dos construtores de GregorianCalendar.

```
Calendar agora = Calendar.gerInstance();
Calendar umaData = new GregorianCalendar(2010, Calendar.MAY, 20);
Calendar outraData = new GregorianCalendar();
```

A seguir são apresentados os métodos mais utilizados das classes Calendar e GregorianCalendar.

add()

Este método recebe como argumento o campo de Calendar a ser alterado e um valor representando a quantidade de tempo a ser adicionado ou removido.

```
public class ExemploCalendar {
	public static void main(String[] args) {
		Calendar c = new GregorianCalendar(2010, Calendar.MAY, 20);
		data.add(Calendar.DAY_OF_MONTH, -12);
		data.add(Calendar.MONTH, 6);

		// Apresentará Mon Nov 08 00:00:00 BRST 2010
		System.out.println(data.getTime());
	}
}
```

after(), before() e equals()

São métodos equivalentes aos da classe Date, exceto que só funcionam com argumentos do tipo Calendar.

```
public class ExemploCalendar {
	public static void main(String[] args) {
		Calendar agora = Calendar.getInstance();
		Calendar data = new GregorianCalendar(2010, Calendar.MAY, 20);

		String st = "A data: " + data.getTime() + " é ";
		String st2 = " a data de hoje";

		if(data.after(agora)) {
			System.out.println(st + "posterior" + st2);
		} else if(data.before(agora)) {
			System.out.println(st + "anterior" + st2);
		} else if(data.equals(agora)) {
			System.out.println(st + "igual" + st2);
		} else {
			System.out.println("Argumento inválido");
		}
	}
}
```

getTime()

Retorna um objeto do tipo Date.

```
Calendar c = new GregorianCalendar(2010, Calendar.MAY, 20);
Date data = c.getTime();
```

getTimeMillis()

Este método é equivalente ao método Date.getTime() e retorna o número do tipo long de milissegundos referente a diferença entre a data representada por este objeto Calendar e 1/1/1970 00:00:00h.

Classe Instant

Esta classe modela um único ponto instantâneo na linha do tempo. Isso pode ser usado para registrar os carimbos de hora do evento no aplicativo.

O intervalo de um instante requer o armazenamento de um número maior que um `long`. Para conseguir isso, a classe armazena um long representando os segundos para época e um `int` representando nanossegundo de segundo, que sempre estará entre 0 e 999.999.999. Os períodos-segundos são medidos a partir da época Java padrão de 1970-01-01T00:00:00Z onde os instantes após a época têm valores positivos e os instantes anteriores têm valores negativos. Para ambas as partes de época e nanossegundo, um valor maior é sempre mais tarde na linha de tempo do que um valor menor.

```
Instant agora = Instant.now();
Instant data = Instant.from(LocalDate.of(2018, Month.NOVEMBER, 23));
```

Classe LocalDate, LocalTime e LocalDateTime

LocalDate é um objeto de data e hora imutável que representa uma data, geralmente visto como ano-mês-dia. Outros campos de data, como dia da semana, dia da semana e semana do ano, também podem ser acessados.

```
LocalDate agora = LocalDate.now();
LocalDate data = LocalDate.of(2018, Month.NOVEMBER, 23);
```

LocalTime é um objeto de data e hora imutável que representa uma hora, geralmente vista como hora-minuto-segundo. O tempo é representado para uma precisão de nanossegundos.

```
LocalTime agora = LocalTime.now();
LocalTime hora = LocalTime.of(10, 54);
LocalTime outraHora = LocalTime.of(8, 22, 45);
```

LocalDateTime é um objeto de data e hora imutável que representa uma data e hora, geralmente visto como ano - mês - dia - hora - minuto - segundo. Outros campos de data e hora, como dia da semana, dia da semana e semana do ano, também podem ser acessados. O tempo é representado para uma precisão de nanossegundos.

```
LocalDateTime agora = LocalDateTime.now();
LocalDateTime data = LocalDateTime.of(2018, Month.NOVEMBER, 23, 10, 54);
```

isAfter(), isBefore() e equals()

São métodos equivalentes aos da classe Date, exceto que só funcionam entre argumentos do mesmo tipo LocalDate, LocalTime ou LocalDateTime.

```
public class ExemploLocalDate {
	public static void main(String[] args) {
		LocalDate agora = LocalDate.now();
		LocalDate data = LocalDate.of(2010, Month.MAY, 20);
```

```
            String st = “A data: “ + data.getTime() + “ é “;
            String st2 = “ a data de hoje”;

            if(data.isAfter(agora)) {
                  System.out.println(st + “posterior” + st2);
            } else if(data.isBefore(agora)) {
                  System.out.println(st + “anterior” + st2);
            } else if(data.equals(agora)) {
                  System.out.println(st + “igual” + st2);
            } else {
                  System.out.println(“Argumento inválido”);
            }
      }
}
```

Formatações

Classe DateFormat

A classe DateFormat permite a conversão de uma string com informações sobre uma date em um objeto do tipo Date e também permite apresentar a data com diferentes formatações, dependendo das necessidades de utilização, tornando sua visualização mais agradável aos usuários.

Ao criar um objeto a partir da classe DateFormat, ele conterá informação a respeito de um formato particular no qual a data será apresentada. Existem diferentes formatos aceitos para a classe DateFormat apresentados na tabela a seguir.

Nome do Formato	Exemplo
default	10/12/2010
DateFormat.SHORT	10/12/10
DateFormat.MEDIUM	10/12/2010
DateFormat.LONG	10 de Dezembro de 2010
DateFormat.FULL	Sexta-feira, 10 de Dezembro de 2010

format()

Método utilizado na conversão de um objeto do tipo Date para String.

```
DateFormat df = DateFormat.getDateInstance(DateFormat.FULL);
Date hoje = new Date();
String data = df.format(hoje);

System.out.println(data);
```

parse()

Este método é utilizado para converter uma String contento uma data em um objeto do tipo Date. Sempre que utilizar este método será necessário o tratamento da exceção ParseException.

```
public class ExemploDateFormat {
	public static void main(String[] args) {
		try {
			DateFormat df = DateFormat.getDateInstance();
			String data = "15/01/2011";
			Date hoje = df.parse(data);

			System.out.println(hoje);
		} catch(ParseException ex) {
```

```
                System.out.println("Data Inválida!");
            }
        }
}
```

Classe SimpleDateFormat

Ela permite criar formatos alternativos para a formatação de datas e horas, dependendo das necessidades do desenvolvedor, ou seja, essa classe possibilita expandir as capacidades da classe DataFormat.

Para que o seja criado o formato de data/hora deve ser montada uma String utilizando os caracteres apresentados na tabela a seguir.

Caractere	Descrição	Exemplo
G	Designador da era	AD
y	Ano	2005 ou 05
M	Mês do ano	Jul ou 07
w	Semana do ano	15
W	Semana do mês	3
D	Dia do ano	234
d	Dia do mês	5
F	Dia da semana no mês	2
E	Dia da semana	Sex
a	AM ou PM	AM
H	Hora do dia (0-23)	0
k	Hora do dia (1-24)	23

Caractere	Descrição	Exemplo
K	Hora do dia (0-11)	2
h	Hora do dia (1-12)	5
m	Minuto da hora	10
s	Segundos do minuto	30
S	Milissegundos	978

As letras apresentadas na tabela acima permitem criar os mais diversos tipos de formatação, sejam numéricos ou textuais. Dependendo da combinação de letras e das quantidades de cada letra usada, são criados os mais diversos formatos. Observe que uma simples alteração de minúscula para maiúscula pode gerar um resultado totalmente diferente.

O padrão para a formatação pode ser informado via construtor da classe SimpleDateFormat ou via o método applyPattern().

Para converter String em objeto do tipo Date ou de objeto do tipo Date para String basta utilizar os métodos parse() e format() respectivamente. Eles são equivalentes aos métodos da classe DateFormat.

```
public class ExemploSimpleDateFormat {
	public static void main(String[] args) {
		try {
			SimpleDateFormat df = new SimpleDateFormat();
			df.applyPattern(“dd/MM/yyyy“);

			String data = “15/01/2011“;
			Date hoje = df.parse(data);

			df.applyPattern(“EEEE, ‘dia’ d ‘de’ MMMM ‘de’ yyyy“);
			System.out.println(df.format(hoje));
		} catch(ParseException ex) {
			System.out.println(“Data Inválida!”);
		}
	}
}
```

Classe DateTimeFormatter

A classe DateTimeFormat permite a formatação de objetos LocalDate, LocalTime e LocalDateTime com diversos estilos fornecidos pela classe FormatStyle.

```
LocalDate agora = LocalDate.now();
String data = agora.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL);

System.out.println(data);
```

Outro recurso de formatação oferecido pela classe DateTimeFormatter é a formatação através da utilização de uma String utilizando os caracteres utilizados na classe *SimpleDateFormat*.

```
LocalDate agora = LocalDate.now();
String data = agora.format(DateTimeFormatter.ofPattern(“dd/MM/yyyy”);

System.out.println(data);
```

Classe FormatStyle

A enumeração FormatStyle oferece um conjunto de estilos para a formatação de data e hora para as classe LocalDate, LocalTime e LocalDateTime utilizando DateTimeFormatter. Os possíveis formatos aceitos estão apresentados na tabela a seguir.

Nome do Formato	Exemplo Data	Exemplo Hora
FormatStyle.SHORT	23/11/2018	12:31
FormatStyle.MEDIUM	23 de nov de 2018	12:31:02
FormatStyle.LONG	23 de novembro de 2018	—
FormatStyle.FULL	sexta-feira, 23 de novembro de 2018	—

```
public class ExemploDateFormatter {
	public static void main(String[] args) {
		try {
			DateTimeFormatter fmtSimple =
				DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
			DateTimeFormatter fmtCompleto = DateTimeFormatter.ofPattern(
				“EEEE, ‘dia’ d ‘de’ MMMM ‘de’ yyyy“);

			String data = “23/11/2018“;
			LocalDate hoje = LocalDate.parse(data, fmtSimples);
			System.out.println(hoje.format(fmtCompleto));
		} catch(DateTimeParseException ex) {
			System.out.println(“Data Inválida!”);
		}
	}
}
```

Classe NumberFormat

A classe NumberFormat é a classe abstrata base para todos as formatações numéricas. Esta classe fornece a interface para a formatação e interpretação de números.

Use `getInstance()` ou `getNumberInstance()` para obter o formatador padrão para números. Use `getIntegerInstance()` para formação de inteiros. O método `getCurrencyInstance()` para formatação de valores monetários. E `getPercentInstance()` para formatação de percentagem.

Classe DecimalFormat

DecimalFormat é a subclasse concreta de NumberFormat que formata números decimais. Ela têm uma variedade de recursos para suportar diferentes tipos de números incluindo inteiros, números com ponto flutuante, notação científica, percentagem e valores monetários.

A classe DecimalFormat utiliza um conjunto de padrões e símbolos para especificar sua formatação, conforme tabela abaixo:

Símbolo	Significado
0	Dígitos
#	Dígitos, na ausência é apresentada zero
.	Separador decimal
-	Sinal negativo
,	Separador de centena
E	Separa a mantissa e expoente na notação científica
;	Separa os padrões para valores positivos e negativos
%	Multiplica por 100
\u2030	Multiplica por 1000
\u00a4	É substituído pelo simbolo monetário
'	Utilizado para marcar os textos que não devem ser considerados como padrão

```
public class ExemploDecimalFormat {
	public static void main(String[] args) {
		try {
			DecimalFormat df = new DecimalFormat("#,##0.00;-#,##0.00");

			String s = "1.654,1";
			Number num = df.parse(s);
			double d = num.doubleValeu() * -1;
			df.applyPattern("\u00a4 #,##0.00;\u00a4 (#,##0.00)");

			System.out.println(df.format(d)); //Apresentará R$ (1.654,10)
		} catch(DateTimeParseException ex) {
			System.out.println("Valor Inválido!");
		}
	}
}
```

Classe Formatter

Esta classe oferece suporte a leiautes justificados e alinhados, formatação para números, Strings, data e hora para uma localidade (Locale) específica. Tipos Java tais como, byte, BigDecimal, e Calendar são suportados.

```
// Exemplo de Formatação numérica
Formatter fmt = new Formatter();
double num = -1654.1;

// Apresentará R$ (1.654,10)
System.out.println(fmt.format("R$ %(,.2f\n", num));

// Exemplo de Formatação de Data
Date data = new Date();

// Apresentará 24/11/2010
System.out.println(fmt.format("%1$te/%1$tm/%1$tY\n", data));

// Apresentará Quarta-feira, dia 11 de Novembro de 2010
System.out.println(fmt.format("%1$tA, dia %1$tm de %1$tB de %1$tY", data));
```

A formatação a ser aplicada é especificada por uma String de formatação. A String de formatação contém uma sequencia de mnemônicos iniciados com o caractere"%".

A sequencia de mnemônicos para formatar caracteres e tipos numéricos têm o seguinte formato:

%[*índice do argumento$*][*indicador*][*tamanho*][.*precisão*]*conversão*

A sequencia de mnemônicos para formatar data e hora têm o seguinte formato:

%[*índice do argumento$*][*indicador*][*tamanho*]*conversão*

O índice do argumento é opcional e indica a posição do argumento a ser formatado.

O indicador é opcional e é um caractere que modifica a formatação.

O tamanho é um número não negativo que indica a quantidade mínima de caracteres a ser utilizada na formatação.

A precisão é opcional e indica através de um número não negativo a quantidade de dígitos apresentada na posição depois da vírgula.

A conversão é um caractere obrigatório que determina o tipo da formatação aplicada.

Abaixo é apresentada uma tabela com os caracteres mais utilizados na conversão de caracteres, tipos numéricos, datas e horas.

Conversão	Descrição
s ou S	Invoca toString() do argumento
c ou C	É aplicado em caracteres
d	É utilizado em números inteiros
e ou E	É utilizado em números de ponto flutuante e apresenta notação científica
f	É utilizado em números de ponto flutuante
t ou T	É utilizado como prefixo em formatação de data e hora
n	Insere uma nova linha na formatação

A tabela a seguir têm relacionado os subtipos para a formatação de data e hora.

Conversão	Descrição
H	Hora do dia na faixa. 00-23
I (i maiúsculo)	Hora do dia no formato 12 horas na faixa 01-12
k	Hora do dia na faixa 0-23, sem acréscimo de zeros adicionais
l (L minúsculo)	Hora do dia no formato 12 horas na faixa 1-12, sem acréscimo de zeros adicionais
M	Minuto na hora ex. 00 – 59
S	Segundos no minuto ex. 00-60
L	Milissegundos ex. 000- 999
p	Marcados am ou pm
B	Nome do mês ex. Janeiro
b	Nome do mês abreviado ex. Jan
A	Nome da semana ex. Domingo
a	Nome da semana abreviado ex. Dom
Y	Ano com quatro dígitos ex. 1999

Conversão	Descrição
y	Ano com dois dígitos ex. 99
m	Número do mês com 2 dígitos ex. 02
d	Dia do mês com 2 dígitos ex. 01 – 31
e	Dia do mês ex 1 – 31

A tabela a seguir apresenta os indicadores utilizado em formatações

Conversão	Descrição
-	Alinha o resultado à esquerda
+	Sempre apresentará sinal
(espaço em branco)	Acrescentará espaço em valores positivos
0 (número zero)	Preenchimento com zeros em números
,	Apresentará ponto nas centenas
(	Circundará com parenteses números negativos

As classes String e OutputStream também implementam a formatação, tornando possível a utilização do método format diretamente.

```
// Apresentará 24/11/2010
System.out.format("%1$te/%1$tm/%1$tY%n", new Date());
```

Classe Pattern

Uma representação compilada de uma expressão regular.

Uma expressão regular descrita como uma String, deve ser compilada numa instancia desta classe. O padrão resultante pode ser utilizado para criar um objeto do tipo Matcher que identifica o padrão compilado em uma sequencia de caracteres arbitrário.

O método matches() oferece a conveniência de processar uma expressão regular diretamente. Este método compila a expressão regular e efetua a identificação do padrão resultante contra a String fornecida.

```
boolean b = Pattern.matches("a*b", “aaaaab");
```

A tabela abaixo apresenta os construtores para os padrões regulares mais usuais.

Construtor	Identifica
[abc]	a, b ou c
[^abc]	Qualquer caractere exceto a, b e c
[a-z]	Todos os caracteres de a até z
[a-zA-Z]	Todos os caracteres de a até z e de A até Z

Construtor	Identifica
.	Qualquer caractere
\d	Dígitos de 0 até 9
\w	Equivalente [a-zA-Z_0-9]
X?	Ocorrência de X, uma ou nenhuma vez
X*	Ocorrência de X, zero ou mais vezes
X+	Ocorrência de X, uma ou mais vezes
X{n}	Ocorrência de X, exatamente n vezes
X{n,}	Ocorrência de X, pelo menos n vezes
X{n,m}	Ocorrência de X, pelo menos n vezes mas não mais que m
XY	X seguido de Y
X|Y	Ou X ou Y

Abaixo temos um exemplo da utilização de expressões regulares para validar o código do produto no formato “9999-99”:

```
String st = JOptionPane.showInputDialog("Informe o Cod. Produto");

if(st != null && st.matches(“[0-9]{4}-[0-9]{1,2}”)) {
	System.out.println("Cod. Produto válido!");
} else {
	System.out.println("Cod. Produto inválido!”);
}
```

Enumerações (enum)

A partir da versão 5.0 do JDK é suportado tipos enumerados conhecidos como enum. Na sua forma mais simples as enumerações se parecem como uma lista de valores:

```
public enum Estacoes {
	PRIMAVERA,
	VERAO,
	OUTONO,
	INVERNO;
}
```

Sua aparência pode enganar. Em Java enumerações são muito mais poderosas. Elas definem:

- A `enum` define um tipo da mesma forma que `class` define;
- Limita os valores possíveis para este tipo criando um namespace que agrupa todos os valores possíveis para este tipo;
- As enumerações são definidas como constantes impedindo alterações e possibilitando a uniformidade na sua utilização;
- A impressão de uma enumeração resulta em uma `String` possibilitando que possamos converter enumerações para String com `toString()` e String para enumerações com `valueOf()`;
- Uma enumeração implementa `Comparable` e `Serializable`.

Podemos adicionar comportamento a enumerações, como no exemplo abaixo:

```
public enum Operacao {
	SOMA {
		public double execute(double x, double y) {
			return x + y;
		}
	},
	SUBTRAI {
		public double execute(double x, double y) {
			return x - y;
		}
	},
	MULTIPLICA {
		public double execute(double x, double y) {
			return x * y;
		}
```

```
	},
	DIVIDE {
		public double execute(double x, double y) {
			return x / y;
		}
	};

	public abstract double execute(double x, double y);
}
```

Cada enumeração do tipo `Operacao` tem seu próprio comportamento, mas compartilham o mesmo nome. Assim `SOMA.execute()` efetua a soma de dois double e `DIVIDE.execute()` divide o valor de dois double. Conforme exemplo abaixo:

```
double x = 4;
double y = 2;
// Apresentará os valores abaixo
// 4,000000 SOMA 2,000000 = 6,000000
// 4,000000 SUBTRACAO 2,000000 = 2,000000
// 4,000000 MULTIPLICACAO 2,000000 = 8,000000
// 4,000000 DIVISAO 2,000000 = 2,000000
for (Operacao op : Operacao.values()) {
	System.out.printf("%f %s %f = %f%n", x, op, y, op.execute(x, y));
}
```

Toda a enumeração tem definido automaticamente os métodos:

- equals()-compara duas enumerações e retorna true ou false;
- toString()-retorna uma String representando a enumeração;
- valueOf()-converte uma String em enumeração;
- values()-retorna um array das enumerações declaradas (vide exemplo acima).

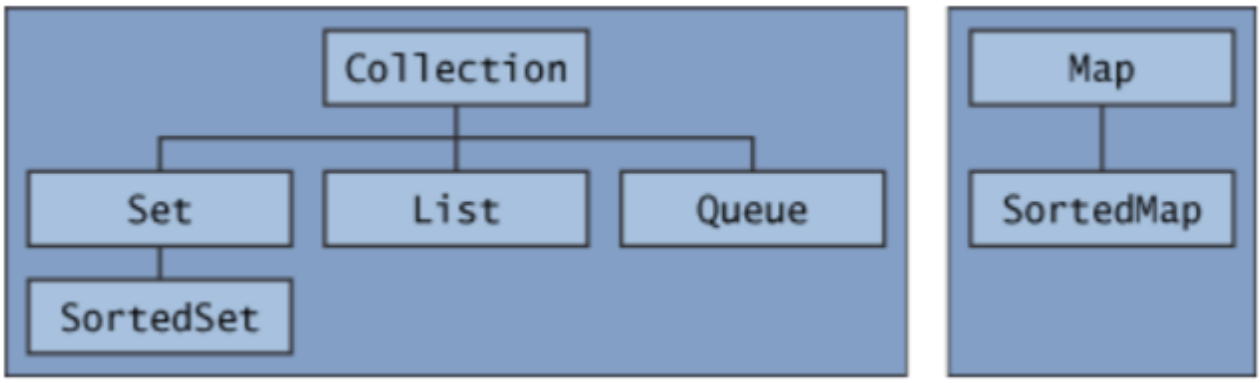
Utiliza enumerações sempre que necessitar de um conjunto fixo de constantes, tais como, sexo, tipos de telefone, estados do pais, por exemplo.

Coleções

O Java Collections Framework é a composição é uma arquitetura unificada para representar e manipular coleções e contém:

- Interfaces: Estas são abstrações de tipos de dados que representam coleções. As interfaces permitem que as coleções sejam manipuladas independentemente dos detalhes de sua implementação;
- Implementações: São as implementações concretas das interfaces. Em essência são estruturas de dados reutilizáveis;
- Algoritmos: São métodos que executam operações úteis, tais como pesquisa e ordenação, em objetos que implementam as interfaces de coleções.

As coleções encapsulam diferentes tipos de coleções, exibidas na figura abaixo:



Estas interfaces permitem que as coleções sejam manipuladas independentemente dos detalhes de suas implementações. As interfaces são a base do Java Collections Framework.

Um Set é um tipo especial de Collection, um SortedSet é um tipo especial de Set, assim por diante. Note também que a hierarquia consiste em duas arvores distintas. Um Map não é uma Collection.

Ainda todas as interfaces de coleções são genéricas. Veja abaixo a declaração da interface Collection:

```
public interface Collection<E> ...
```

A sintaxe "<E>" informa que a interface é genérica. Quando você declara uma instância de Collection você pode e deve especificar o tipo do objeto que será contido na coleção. Desta forma é possível que o tipo dos objetos adicionados nesta coleção possam ser verificados no momento da compilação.

Abaixo segue a descrição das principais interfaces de Collection.

Collection

A interface topo da hierarquia de coleções. Uma Collection representa um grupo de objetos chamados de elementos. A interface `Collection` é o último denominados comum para todas as implementações de coleções e é utilizada como argumento quando é necessário o máximo de generalização.

Alguns tipo de coleções permitem duplicidade de elementos, e outros não. Alguns são ordenados e outros não. As mais específicas sub-interfaces são `Set`, `List` e `Queue`. Abaixo segue uma lista de alguns métodos declarados nesta interface:

métodos	Descrição
size()	Quantos elementos estão contidos na coleção
isEmpty()	true se a coleção estiver vazia
contains()	Verifica se um objeto está na coleção
add()	Adiciona um elemento à coleção
remove()	Remove um elemento à coleção
clear()	Remove todos os elementos de uma coleção
toArray()	Retorna um array dos elementos contidos na coleção

Set

Uma coleção que não pode conter duplicidade de elementos. Esta interface modela a abstração matemática de um set, tais como um conjunto de cartas em um jogo de truco, as disciplinas de um curso superior.

As implementações de `Set` são `HashSet`, que armazena os elementos em uma tabela hash, é sua implementação mais rápida, no entanto não garante a ordem dos elementos. `TreeSet`, que armazena os elementos em uma arvore balanceada, os elementos são ordenados com base em seus valores. É mais lento que o `HashSet`. `LinkedHashSet`, que implementa uma tabela hash com uma lista ligada, permitindo ordenação de seus elementos baseado na ordem de inserção.

```
// Declara um set de Strings
Set<String> lista = new TreeSet<String>();

// Inclui objetos no set
lista.add(“abc");
lista.add(“def");
lista.add(“fgh");

// Ordena o set
// A implementação TreeSet ordena automaticamente na
```

```
// inserção e não permite duplicidade de chave

// Apresenta o set com foreach
for (String txt : lista)
      System.out.println(txt);

// Testa da existencia do objeto no set
if(lista.contains(“def"))
      // remove o objeto do set
      lista.remove(“def");

// Apresenta o set com while
Iterator<String> it = lista.iterator();
while(it.hasNext()) {
      String txt = it.next();
      System.out.println(txt);
}

// Apresenta o set com stream
lista.stream().forEach(System.out::println);
```

List

Uma coleção ordenada (as vezes chamada de sequencia). `List` pode conter elementos duplicados. Ao adicionarmos um elemento em `List` é associado um índice a este elemento (sua posição). As implementações mais utilizadas de `List` são `ArrayList` e `Vector`.

```
// Declara um list de Strings
List<String> lista = new ArrayList<String>();

// Inclui objetos na lista
lista.add(“abc");
lista.add(“def");
lista.add(“fgh”);

// Ordena a lista
Collections.sort(lista);

// Apresenta a lista com foreach
for (String txt : lista)
      System.out.println(txt);
```

```
// Testa da existencia do objeto na lista
if(lista.contains(“def"))
      // remove o objeto da lista
      lista.remove(“def”);

// Apresenta a lista com for
for (int i = 0; i < lista.size(); i++) {
      String txt = lista.get(i);
      System.out.println(txt);
}

// Apresenta a lista com stream
lista.stream().forEach(System.out::println);
```

Queue

É uma coleção que armazena elementos em uma FIFO (primeiro que entra é o último que sai). Numa file FIFO, todos os novos elementos são adicionados ao final da fila. Abaixo seguem alguns métodos de `Queue`:

métodos	Descrição
remove() e poll()	Remove e retorna o elemento no topo da fila
element() e peek()	Retorna o elemento no topo da fila sem removê-lo
offer()	Usado somente em Queues com limites, adiciona elementos

```
// Declara um queue de Strings
Queue<String> lista = new PriorityQueue<String>();

// Inclui objetos no queue
lista.add(“abc");
lista.add(“def");
lista.add("fgh");

// Ordena o map
// A implementação PriorityQueue ordena automaticamente na
// inserção

// Apresenta ao queue com foreach
for (String txt : lista)
      System.out.println(txt);
```

```
// Testa da existencia do objeto no map pela chave
if(lista.contains(“def"))
	// remove o objeto do map pela chave
	lista.remove(“def");

// Apresenta o queue com while
Iterator<String> it = lista.iterator();
while(it.hasNext()) {
	String txt = it.next();
	System.out.println(txt);
}

// Apresenta a lista com stream
lista.stream().forEach(System.out::println);

// Retirando os objetos do queue
for (int i = lista.size(); i > 0; i--) {
	System.out.println(lista.poll());
}
```

Map

É um objeto que associa valores a chaves. Um Map não pode conter chaves duplicadas: Cada chave só pode estar associada a um valor. Abaixo seguem alguns métodos de map:

métodos	Descrição
put()	Adiciona chaves e valores ao Map
get()	Retorna o elemento associado a chave informada
containsKey()	Retorna true se a chave existe no Map
containsValue()	Retorna true se o valor existe no Map
values()	Retorna uma Collection dos elementos contidos no Map

```
// Declara um map de Strings
Map<String, String> lista = new TreeMap<String, String>();

// Inclui objetos no map
lista.put("chave1", “abc");
lista.put("chave2", “def");
lista.put("chave3", "fgh");

// Ordena o map
```

```
// A implementação TreeMap ordena automaticamente na
// inserção pela chave e não permite duplicidade desta e
// substitui o valor caso isto aconteça

// Apresenta o map com foreach
for (String txt : lista.values())
      System.out.println(txt);

// Testa da existencia do objeto no map pela chave
if(lista.containsKey(“chave2"))
      // remove o objeto do map pela chave
      lista.remove(“chave2");

// Apresenta o map com while
Iterator<String> it = lista.values().iterator();
while(it.hasNext()) {
      String txt = it.next();
      System.out.println(txt);
}

// Apresenta o map com stream
lista.values().stream().forEach(System.out::println);
```

Operações com Arquivos

Antes de falar a respeito de como efetuar gravações de arquivos em Java é necessária a definição do que é Streams.

Streams

Streams são uma abstração de baixo nível para a comunicação de dados em Java. Um Stream representa um ponto num canal de comunicação.

O canal de comunicação normalmente conecta um output stream a um correspondente input stream. Tudo o que é gravado no output stream pode ser lido do input stream. Esta conexão pode ocorrer a partir de um ponto de conexão através de uma rede de computadores, de uma área de memória entre processos, para um arquivo, ou para a console do computador. Os Streams oferecem uma interface padronizado para a transferência de dados para aplicações, independente de qual seja o canal de comunicação utilizado.

Todas as classes que efetuam operações de leitura e escrita (I/O) em Java pertencem ao pakage "java.io".

FIFO – First in First out (O primeiro a entrar é o primeiro a sair)

Os Streams são uma fila do tipo FIFO. Isto significa que a primeira coisa que for escrita em um output stream será a primeira a ser lida o input stream.

Assim, Streams oferecem um acesso sequencial para os canais de comunicação. Muitas implementações de Streams não oferecem acesso randômico em função do canal de comunicação utilizado.

Uma importante característica dos Streams que que existe o bloqueio do processamento quando é tentada a leitura de um input stream enquanto não existe nada a ser lido. Assim a única forma da liberação do bloqueio é o envio de uma informação através do output stream corresponde.

Classe File

Esta classe representa o nome de um arquivo independente da arquitetura do Sistema Operacional. Ela oferece vários métodos para determinar informações sobre arquivos e diretórios, bem como permitindo a modificação de seus atributos.

Para obtermos uma instância de objeto File utilizamos um dos seguintes construtores:

```
File f1 = new File("c:\\MeusProjetos");
File f2 = new File("c:\\Users\Fulano\\Desktop", "meuDoc.txt");
File f3 = new File(f1, "Projeto1.doc");
```

Abaixo segue a tabela com alguns dos métodos de File:

Método	Descrição
canRead()	Retorna true se o arquivo pode ser lido
canWrite()	Retorna true se o arquivo pode ser gravado
delete()	Retorna true se obtiver sucesso na deleção do arquivo
exists()	Retorna true se o arquivo existir
getName()	Retorna o nome do arquivo sem o diretório
getPath()	Retorna o nome do arquivo com a parte do diretório
getDirectory()	Retorna o nome do diretório onde o arquivo reside
isFile()	Retorna true se for um arquivo
isDirectory()	Retorna true se for um diretório
length()	Retorna o tamanho do arquivo
list()	Retorna um array de Strings contendo os nomes dos arquivos e sub-diretórios contidos num diretório
mkdir()	Retorna true se conseguir criar o diretório
renameTo()	Retorna true se conseguir renomear o arquivo

A seguir um exemplo de um trecho de código que teste pela existência de um arquivo, criando se necessário, testando se pode gravar e se é um diretório e renomeando um arquivo:

```
if(!f1.exists()) {
      f1.mkdir();
} else if(f1.canWrite() && f1.isDirectory()) {
      f3.renameTo(f2);
}
```

Classes FileOutputStream, FileWriter, PrintWriter, FileInputStream, FileReader e BufferedInputStream

As classes FileOutputStream, FileWriter e PrintWriter são utilizadas para a gravação de dados em arquivos. Enquanto FileOutputStream é orientado a byte e oferece os métodos `write(byte[] b, ...)`, FileWriter e PrintWriter são orientados a caractere e oferecem os métodos `write(String s, ...)`.

PrintWriter ainda acrescenta métodos tais como `printf()`, `print()` e `format()` semelhantes aos utilizados em `System.out`.

Abaixo é apresentado um exemplo de utilização das classes FileOutputStream, FileWriter e PrintWriter, vale notar a necessidade de tratamento de exceção do tipo FileNotFoundException e IOException.

```
try {
      // Declara um objeto do tipo File
      File fl = new File(“c:/User/Fulano/Desktop/meuDoc.txt");

      // Abre o arquivo para gravação
      FileOutputStream fo = new FileOutputStream(fl);

      // Grava um texto no arquivo
      String txt = "Texto de exemplo 1\n”;

      // Transforma a String em array de bytes fo.write(txt.getBytes());
      // Fecha o arquivo
      fo.close();

      // Abre o arquivo para acrecentar mais textos
      FileWriter fw = new FileWriter(fl, true);

      // Grava um texto
      fw.write("Texto de Exemplo 2\n”);

      // Cria um PrintWriter a partir do FileWriter
      PrintWriter pw = new PrintWriter(fw);

      // Grava um novo texto
      pw.println("Texto de Exemplo 3”);

      // Fecha o arquivo
      fw.close();
} catch(FileNotFoundException ex) {
      ex.printStackTrace();
} catch(IOException ex) {
      ex.printStackTrace();
}
```

As classes FileInputStream, FileReader e BufferedReader são utilizadas para a leitura de dados de arquivos. Enquanto FileInputStream é orientado a byte e oferece o método `read(byte[] b, ...)`, FileReader e BufferedReader são orientados a caractere. Enquanto FileReader oferece o método `write(char[] c, ...)`, BufferedReader oferece o método readLine() que é mais eficiente para ler linhas de arquivos texto.

Abaixo é apresentado um exemplo de utilização das classes FileInputStream, FileReader e BufferedReader, vale notar a necessidade de tratamento de exceção do tipo FileNotFoundException e IOException.

```
try {
	// Declara um objeto do tipo File
	File fl = new File(“c:/User/Fulano/Desktop/minhaImagem.dat");

	// Abre o arquivo para leitura
	FileInputStream fi = new FileInputStream(fl);

	// Reserva a área de leitura
	byte[] buffer = new byte[1024];

	// Lê 1024 bytes do arquivo
	fi.read(buffer);

	// Fecha o arquivo
	fi.close();

	// Abre o arquivo para leitura
	FileReader fr = new FileReader(“C:/meusDados/lista.txt");

	// Reserva a área de leitura
	char[] texto = new char[1024];

	// Lê uma sequencia de caracteres do arquivo
	fr.read(texto);

	// Cria um PrintWriter a partir do FileWriter
	BufferedReader br = new BufferedReader(fr);

	// Lê todas as linhas restante do arquivo
	String linha = br.readLine();
	while(linha != null) {
		System.out.println(linha); linha = br.readLine();
	}
```

```
        // Fecha o arquivo
        fr.close();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

Gravação e Leitura de Objetos

A classe ObjectOutputStream é utilizada para gravar objetos em arquivos, este processo é chamado de persistência de objetos. O que ocorre com o objeto é sua serialização, ou seja sua transformação em uma sequencia de bytes para posterior armazenamento.

Toda a informação a respeito da classe que o objeto pertence, sua herança e agregações também é armazenada.

Um objeto para que possa ser serializado deve implementar a interface Serializable obrigatoriamente.

Gravamos um objeto utilizando o método `writeObject()` de ObjectOutputStream, conforme exemplo a seguir.

```
try {
        // Cria um objeto para ser gravado
        List<String> lista = new ArrayList<String>();

        // Adiciona objetos na lista
        lista.add(“Texto");
        lista.add(“java");
        lista.add("Novo");

        // Cria um arquivo para gravação
        FileOutputStream fo = new FileOutputStream(“meusObjetos.dat");

        // Cria um ObjectOutputStream a partir do FileOutputStream
        ObjectOutputStream objOut = new ObjectOutputStream(fo);

        // Grava o objeto
        objOut.writeObject(lista);
```

```
        // fecha o arquivo
        objOut.close();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

A classe ObjectInputStream é utilizada para ler objetos armazenados em arquivos. O que ocorre com os bytes lidos é a reconstrução para o objeto de origem.

Toda a informação a respeito da classe que o objeto pertence, sua herança e agregações é utilizada para a reconstrução dos objetos.

Lemos um objeto utilizando o método `readObject()` de ObjectInputStream, conforme exemplo a seguir.

```
try {
        // Abre um arquivo para leitura
        FileInputStream fi = new FileInputStream(“meusObjetos.dat");

        // Cria um ObjectInputStream a partir do FileInputStream
        ObjectInputStream objIn = new ObjectInputStream(fi);

        // Lê o objeto
        List<String> lista = (List<String>)objIn.readObject();

        // fecha o arquivo
        objIn.close();

        // Exibe os objetos lidos
        for(String txt : lista) {
                System.out.println(txt);
        }
} catch(ClassNotFoundException ex) {
        ex.printStackTrace();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

Note que quando utilizamos `readObject()` necessitamos de um cast (List<String> - do exemplo) e que também tratamos a exceção ClassNotFoundException.

Conectividade de Banco de Dados: JDBC

O que é JDBC

No verão de 1996, a Sun lançou a primeira versão do kit JDBC (Java Database Connectivity, conectividade a banco de dados Java). Esse pacote permite aos programadores se conectarem com um banco de dados, consulta-lo ou atualiza-lo, usando o SQL (Structured Query Language, linguagem de consultas estruturada). (A linguagem SQL é um padrão da área de informática para acesso a banco de dados.) Achamos que esse foi um dos desenvolvimentos mais importantes na programação para a plataforma Java. Não é apenas que os banco de dados estejam entre os usos mais comuns de hardware e software atualmente. Afinal, existem muitos produtos nesse mercado; então, por que achamos que a linguagem de programação Java tem potencial para fazer um grande papel? O motivo da linguagem Java e do JDBC terem uma vantagem fundamental sobre outros ambientes de programação de banco de dados é o seguinte:

- Os programas desenvolvidos com a linguagem de programação Java e JDBC são independentes de plataforma e de fornecedor.

O mesmo programa de banco de dados, escrito em Java, pode ser executado em um computador NT, um servidor Solaris ou um aplicativo de banco de dados utilizado na plataforma Java. Você pode mover seus dados de um banco de dados para outro; por exemplo, do Microsoft SQL Server para Oracle, ou mesmo para um pequeno banco de dados incorporado em um aparelho eletrônico, e o mesmo programa ainda poderá ler seus dados. Isso contrasta fortemente com a programação de banco de dados tradicional. É também muito comum alguém escrever aplicativos de banco de dados em uma linguagem de banco de dados proprietária, usando um sistema de gerenciamento de banco de dados que esteja disponível apenas em um fornecedor. O resultado é que você pode executar o aplicativo resultante apenas em uma ou duas plataformas.

Como parte do lançamento da linguagem Java 2, em 1998, uma segunda versão da JDBC também foi publicada.

O Projeto do JDBC

Desde o início, os desenvolvedores da tecnologia Java da Sun sabiam do potencial que a linguagem Java mostrava para se trabalhar com banco de dados. A partir de 1995, eles começaram a trabalhar na extensão da biblioteca Java padrão para lidar com o acesso via SQL aos banco de dados. O que eles esperavam fazer primeiro era estender a linguagem Java de modo que ela pudesse se comunicar com qualquer banco de dados de acesso aleatório, usando Java "puro". Não demorou muito para que eles percebessem que isso era uma tarefa impossível: havia simplesmente bancos de dados demais no mercado, usando protocolos demais. Além disso, embora os fornecedores de banco de dados estivessem todos favoráveis à Sun no fornecimento de um protocolo de rede padrão para acesso a banco de dados, eles só estariam a favor se a Sun decidisse usar o protocolo de rede deles.

O que todos os fornecedores de banco de dados concordavam era que seria interessante se a Sun fornecesse uma API Java pura para acesso com SQL junto com um gerenciador de drivers para permitir que drivers de outros fornecedores se conectassem a banco de dados específicos. Os fabricantes de banco de dados poderiam fornecer seus próprios drivers para se ligar ao gerenciados de drivers. Haveria então um mecanismo simples para registrar drivers de terceiros junto ao gerenciador de driver – sendo o objetivo de que os drivers precisassem apenas seguir os requisitos apresentados na API do gerenciador de driver.

Após um longo período de discussão pública, a API para acesso a banco de dados se tornou a API JDBC e as regras para a escrita de drivers foram encapsuladas na API de driver JDBC. (A API de drivers JDBC é de interesse apenas para os fornecedores de banco de dados e provedores de ferramentas de banco de dados.)

Esse protocolo segue o modelo ODBC da Microsoft de muito sucesso, que forneceu uma interface da linguagem de programação C para acesso a banco de dados. O JDBC e o ODBC têm por base a mesma idéia: Os programas escritos de acordo com a API JDBC se comunicariam com o gerenciador de drivers JDBC que, por sua vez, usaria os drivers que estivessem ligados a ele nesse momento, para falar com o banco de dados real.

Mais precisamente, o JDBC consiste de duas camadas. A camada superior é a API JDBC. Essa API se comunica com a API de driver gerenciador JDBC, enviando para ela as diversas instruções SQL. O gerenciador deve (de forma transparente para o programador) se comunicar com os vários drivers de terceiros que efetivamente se conectam com o banco de dados, e retornar as informações da consulta ou executar a ação especificada por ela.

Os drivers JDBC são classificados nos seguintes tipos:

- Um driver tipo 1 transforma JDBC em ODBC e emprega com um driver ODBC para se comunicar com o banco de dados. A Sun inclui um driver assim, a ponte JDBC/ODBC (JDBC/ODBC Bridge), no JDK. Entretanto, a ponte não oferece suporte a JDBC2 e exige a distribuição e a configuração correta de um driver ODBC. A ponte é útil para testes, mas não a recomendamos para uso em produção.

- Um driver tipo 2 é um driver parcialmente escrito na linguagem de programação Java e parcialmente em código nativo, que se comunica com a API cliente de um banco de dados. Quando você usa tal driver, deve instalar algum código específico da plataforma, além de uma biblioteca Java.

- Um driver tipo 3 é uma biblioteca cliente Java pura que usa um protocolo independente de banco de dados para comunicar pedidos do banco de dados para um componente servidor, o qual transforma os pedidos em um protocolo específico do banco de dados. A biblioteca cliente é independente do banco de dados real, simplificando assim a distribuição.

- Um driver tipo 4 é uma biblioteca Java pura que transforma pedidos JDBC diretamente em um protocolo específico do banco de dados.

A maioria dos vendedores de banco de dados fornece um driver tipo 3 ou tipo 4 com o banco de dados. Além disso, várias outras empresas são especializadas na produção de drivers com melhor atendimento aos padrões, suporte a mais plataformas ou, em alguns casos, simplesmente melhor confiabilidade do que os drivers fornecidos pelos fabricantes de banco de dados.

Resumindo, o objetivo final do JDBC é tornar possível o seguinte:

- Que os programadores possam escrever aplicativos na linguagem de programação Java para acessar qualquer banco de dados, usando instruções SQL padrão ou mesmo extensões especializadas de SQL, enquanto ainda sigam as convenções da linguagem Java. (Todos os drivers JDBC devem oferecer suporte pelo memos para a versão básica do SQL 92.)

- Que os fabricantes de banco de dados e de ferramentas de banco de dados possam fornece os drivers de baixo nível. Assim, eles podem otimizar seus drivers para seus produtos específicos.

Conceitos Básicos de Programação JDBC

A programação com as classes JDBC não é, conceitualmente, muito diferente da programação com as classes normais da plataforma Java: você constrói objetos a partir das classes JDBC principais, estendendo-as através de herança, se necessário.

URLs de Banco de Dados

Ao se conectar com um banco de dados, você deve especificar a fonte dos dados e talvez precise especificar parâmetros adicionais. Por exemplo, os drivers de protocolo de rede podem precisar de uma porta e os drivers ODBC podem precisar de vários atributos.

Conforme você poderia esperar, o JDBC usa uma sintaxe semelhante ao utilizados nas URLs normais da rede para descrever origens de dados. Aqui estão exemplos da sintaxe:

```
jdbc:mysql://localhost:3306/livraria
```

Esse comando acessaria um esquema em banco de dados MySQL chamada "livraria", usando o Driver JDBC. A sintaxe é:

```
jdbc:nome_do_subprotocolo:outras_informações
```

O sub-protocolo é usado para selecionar o driver específico para conexão com o banco de dados.

O formato do parâmetro "outras informações" depende do sub-protocolo usado. A Sun recomenda que, se você estiver usando um endereço de rede como parte do parâmetro "outras informações", use a convenção de atribuição de nomes URL padrão //nome_host:porta/outros. Por exemplo:

```
jdbc:jdbc://192.168.10.72:3306/livraria?useSSL=false
```

Esta URL permitiria estabelecer uma conexão com o banco de dados "livraria" na porta 3306 de 192.168.10.72, usando o valor de atributo JDBC "useSSL" configurado como "false".

Estabelecendo a Conexão

DriverManager é a classe responsável por selecionar drivers de banco de dados e criar uma nova conexão a banco de dados. Entretanto, antes que o gerenciador de drivers possa ativar um driver, este deve ser registrado.

Existem dois métodos para registrar drivers. Seu programa pode configurar a propriedade de sistema jdbc.drivers como uma lista de drivers. Por exemplo, seu aplicativo pode incluir um arquivo de propriedade com a linha abaixo nas propriedades de sistema:

```
jdbc.drivers=com.mysql.cj.jdbc.Driver
```

Essa estratégia permite aos usuários de seu aplicativo instalar os drivers apropriados simplesmente modificando um arquivo de propriedades.

A propriedade jdbc.drivers contém uma lista de nomes de classe para os drivers que o gerenciador de drivers pode usar. Os nomes são separados por dois pontos.

Você também precisa inserir o código do driver em algum ligar no caminho da classe, para garantir que o aplicativo possa carregar a classe.

Alternativamente, você pode registrar um driver manualmente, carregando sua classe. Por exemplo, para carregar o driver JDBC do MySQL, você usa o comando:

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Após registrar drivers, você abre uma conexão de banco de dados semelhante ao exemplo a seguir:

```
String url = "jdbc:mysql://192.168.10.72/livraria?useSSL=false";
String username = "root";
String password = "root132";
Connection con = DriverManager.getConnection(url, username, password);
```

O gerenciador de drivers tentará encontrar um driver que possa usar o protocolo especificado no URL do banco de dados, fazendo uma interação através dos drivers disponíveis correntemente registrados no gerenciador de drivers.

Para nosso programa exemplo, achamos conveniente usar um arquivo de propriedades para especificar o URL, nome de usuário e senha, além do driver de banco de dados. Um arquivo de propriedades típico tem o seguinte conteúdo:

```
jdbc.drivers=com.mysql.cj.jdbc.Driver
jdbc.url=jdbc:mysql://192.168.10.72/livraria?useSSL=false
jdbc.username=root
jdbc.password=root132
```

Aqui está o código para ler um arquivo de propriedades e abrir a conexão de banco de dados.

```
try ( FileInputStream in = new FileInputStream(filename) ) {
      Properties props = new Properties();
      props.load(in);

      String drivers = props.getProperty("jdbc.drivers");
```

```
        if(drivers != null) {
                System.setProperty("jdbc.drivers", drivers);

                String url = props.getProperty("jdbc.url");
                String username = props.getProperty("jdbc.username");
                String password = props.getProperty("jdbc.password");

                return DriverManager.getConnection(url, username, password);
        } else {
                throw new Exception("Falha ao conectar ao Banco de Dados");
        }
} catch(IOException | SQLException ex) {
        throw new Exception("Falha na inicialização da conexão");
}
```

Note que nós simplesmente não incluímos as propriedades carregadas nas propriedades de sistema:

```
System.setProperties(props); // não fazemos isso
```

Essa chamada poderia sobrescrever inadvertidamente outras propriedades de sistema. Portanto, transferimos apenas a propriedade jdbc.drivers.

O método `getConnection` retorna um objeto connection. Você usa o objeto connection para executar consultas e instruções de ação e efetivar ou retroceder transações.

Executando Comandos de Ação

Para executar um comando SQL, primeiro você prepara um objeto PreparedStatement. O objeto Connection que você obteve da chamada a DriverManager.getConnection pode ser usado para criar objetos PreparedStatement inserindo a instrução que deseja executar a partir do código:

```
PreparedStatement stmt = con.prepareStatement("UPDATE livros " +
                "SET preco = preco - 5.00 WHERE titulo NOT LIKE ?");
```

Os locais onde serão inseridos os argumentos para a instrução SQL deverão ser demarcados com o caracter "?".

Em seguida, você associa os valores aos pontos demarcados na instrução SQL, por exemplo:

```
stmt.setString(1, "%HTML 3%");
```

Então, você chama o método execute da classe PreparedStatement:

```
stmt.execute();
```

Os comandos podem ser ações com INSERT, UPDATE e DELETE, assim como comandos de definição de dados, como CREATE TABLE e DROP TABLE. Entretanto, você não pode usar o método `execute` para executar consultas SELECT.

O método `executeUpdate` retorna uma contagem das linhas afetadas pelo comando SQL. Por exemplo, a chamada a execute Update no exemplo anterior retorna o número de registros de livro cujos preços foram diminuídos de 5.00.

Embora não queiramos nos aprofundar no suporte a transações, queremos mostrar a você como se faz para agrupar um conjunto de instruções para formar uma transação que pode ser efetivada (commit) quando tudo correr bem ou desfeita (rollback) como se nenhum dos comandos tivesse sido executado, se um erro tiver ocorrido em um deles.

O principal motivo para agrupar comandos em transações é a integridade do banco de dados. Por exemplo, suponha que queiramos incluir um novo livro em nosso banco de dados de livros. É importante atualizarmos simultaneamente as tabelas Books, Authors e BooksAuthors. Se a atualização fosse para incluir novos registros nas duas primeira tabelas, mas não na terceira, os livros e autores não seriam correspondidos corretamente.

Se você agrupar atualizações para uma transação, esta tem êxito em sua totalidade e pode ser efetivada ou ela falha em algum lugar intermediário. Nesse caso, você pode desfazer a transação e o banco de dados desfaz automaticamente o efeito de todas as atualizações que ocorreram desde a última transação efetivada. Além disso, é garantido que as consultas relatem apenas o estado efetivado do banco de dados.

Por definição, uma conexão de banco de dados está no modo de efetivação automática (auto-commit) e cada comando SQL é efetivado no banco de dados, assim que é executado. Uma vez efetivado o comando, você não pode desfeito.

Para evitar a configuração corrente do modo de efetivação automática, chame o método getAutoCommit da classe Connection.

Você desativa o modo de efetivação automática com o comando:

```
con.setAutoCommit(false);
```

Agora, você cria um objeto PreparedStatement da maneira normal:

```
PreparedStatement.stmt = con.prepareStatement("INSERT INTO livros “ +
                  “(isbm, titulo, preco, idpublicador) VALUES (?,?,?,?)");
```

Chame `execute` qualquer número de vezes:

```
stmt.setString(1, “978-0134685991”);
stmt.setString(2, “Effective Java 3rd Edition”);
stmt.setDouble(3, 36.55);
stmt.setString(4, “01734”);
stmt.execute();

stmt.setString(1, “978-1119247791”);
stmt.setString(2, “Java All-in-One For Dummies”);
stmt.setDouble(3, 18.49);
stmt.setString(4, “02874”);
stmt.execute();

stmt.setString(1, “978-1259589331”);
stmt.setString(2, “Java: The Complete Reference, Tenth Edition”);
stmt.setDouble(3, 37.85);
stmt.setString(4, “05738”);
stmt.execute();
...
```

Quando todos os comandos tiverem sido executados, chame o método commit:

```
con.commit();
```

Entretanto, se ocorrer algum problema, chame:

```
con.rollback();
```

Dessa forma, todos os comandos até a última efetivação são desfeitos automaticamente, Normalmente, você desfaz uma transação que não ela é interrompida por uma SQLException.

Consultando com JDBC

Para fazer uma consulta, primeiro você cria um objeto PreparedStatement, conforme descrito anteriormente. Após você pode executar uma consulta simples usando o objeto executeQuery da classe PreparedStatement, antes porém, fornecendo os argumentos necessário à sua consulta SQL. Note que você não necessita recriar o objeto PreparedStatement para usar a mesma consulta com outros argumentos, basta informar os novos argumentos e reexecutar o método executeQuery.

É claro que você está interessado no resultado da consulta. O objeto executeQuery retorna um objeto do tipo ResultSet, que você usa para percorrer o resultado, um registro (linha) por vez.

```
PreparedStatement stmt = con.prepareStatement(“SELECT * FROM livros “ +
            "WHERE isbn LIKE ?”);
stmt.setString(1, “978-013%”);
ResultSet rs = stmt.executeQuery();
```

O laço básico para analisar um conjunto de resultado, é semelhante ao seguinte:

```
while(rs.next()) {
      examina uma linha do conjunto de resultado
}
```

Ao inspecionar uma linha específica, você desejará conhecer o conteúdo de cada coluna. Um grande número de métodos “get” fornece essa informação.

```
String isbn = rs.getString(1);
double preco = rs.getDouble(“preco”);
```

Existem um método para cada tipo da linguagem de programação Java, como `getString` e `getDouble`. Cada método “get” tem duas formas, uma que recebe um argumento numérico e outra que recebe um argumento string. Quando fornece um argumento numérico, você se refere à coluna com esse número. Por exemplo, `rs.getString(1)` retorna o valor da primeira coluna da linha corrente.

Quando você fornece um argumento string, se refere à coluna no conjunto de resultado com esse nome. Por exemplo, `rs.getDouble(“preco”)` retorna o valor da coluna de nome *preco*. Usar o argumento numérico é muito pouco eficiente, mas o argumento string torna o código mais fácil de ler e manter.

Cada método “get” fará conversões de tipo razoáveis, quando o tipo do método não coincidir com o tipo da coluna. Por exemplo, a chamada `rs.getString(“preco”);` converte o valor em ponto flutuante da coluna *preco* em uma string.

Tipo de dados SQL	Tipo de dados Java
INTEGER ou INT	int
SMALLINT	short
NUMERIC(m, n), DECIMAL(m, n) ou DEC(m, n)	java.sql.Numeric
FLOAT(n)	double
REAL	float
DOUBLE	double
CHARACTER(n) ou CHAR(n)	String
VARCHAR(n)	String
BOOLEAN	boolean
DATE	java.sql.Date

Tipo de dados SQL	Tipo de dados Java
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array

TIPOS DE DADOS SQL E SEUS TIPOS DE DADOS JAVA CORRESPONDENTES

Tipos SQL Avançados

Além de números, strings e datas, muitos bancos de dados podem armazenar objetos grandes, como imagens ou outros dados. No SQL, os objetos binários grandes são chamados BLOBs e os objetos de caractere grandes são chamados de CLOBs. Os métodos `getBlob` e `getClob` retornam objetos de tipo Blob e Clob. Essas classes têm métodos para recuperar os bytes ou caracteres nos objetos grandes.

Um Array SQL é uma seqüência de valores. Por exemplo, em uma tabela Estudante, você pode ter uma coluna Escores que seja um ARRAY OF INTEGER. O método `getArray` retorna um objeto de tipo java.sql.Array. A interface java.sql.Array possui métodos para buscar os valores do array.

Quando você obtém um blob ou um array de um banco de dados, o conteúdo em si é recuperado do banco de dados apenas quando os valores individuais são solicitados. Esse é um aprimoramento de desempenho útil, pois os dados podem ser bastante volumosos.

Atualizações em Lote

Em uma atualização em lote, uma seqüência de comandos é reunida e enviada como um lote.

Os comandos em um lote podem ser ações como INSERT, UPDATE e DELETE, assim como comandos de definição de dados, como CREATE TABLE e DROP TABLE. Entretanto, você não pode incluir comandos SELECT em um lote, pois a execução de um comando SELECT retorna um conjunto de resultados.

Para executar um lote, primeiro deve ser criado um objeto PreparedStatement com a instrução de INSERT:

```
PreparedStatement stmt = con.prepareStatement(“INSERT INTO publicadores “ +
                “(idpublicador, nome, site) VALUES (?,?,?)”);
```

Desligar o Auto-Commit:

```
con.setAutoCommit(false);
```

Agora deve ser executado o método `addBatch` em vez de `execute`:

```
stmt.setString(1, “01734“);
stmt.setString(2, “Pearson Education Inc”);
stmt.setString(3, “www.pearsoned.com”);
stmt.addBatch();

stmt.setString(1, “02874“);
stmt.setString(2, “John Wiley & Sons, Inc.”);
stmt.setString(3, “www.wiley.com”);
stmt.addBatch();

stmt.setString(1, “05738“);
stmt.setString(2, “McGraw-Hill Education”);
stmt.setString(3, “www.mhprofessional.com”);
stmt.addBatch();
```

Por fim deve ser executado o método `executeBatch` para que todo o lote de comandos acumulados seja despachado para o servidor.

```
stmt.executeBatch();
con.commit();
```

Executando Consultas

Considere a consulta de todos os livros de uma editora em particular, independente do autor. A consulta SQL é:

```
SELECT livros.preco, livros.titulo
      FROM livros, publicadores
      WHERE livros.idpublicador = publicadores.idpublicador
            AND publicadores.nome = ?
```

Em vez de construir um comando de consulta separado sempre que o usuário ativa tal consulta, podemos preparar uma consulta com uma variável hospedeira e usa-la muitas vezes, cada vez fornecendo uma string diferente para a variável. Essa técnica nos fornece uma vantagem de desempenho. Quando o banco de dados executa uma consulta, primeiro ele calcula uma estratégia sobre como executa-la eficientemente. Preparando a consulta e reutilizando-a, você garante que a etapa de planejamento seja feita apenas uma vez.

Cada argumento em uma consulta preparada é indicada com uma "?". Se houver mais de uma variável, então você deverá controlar as posições da "?", ao configurar os valores. Por exemplo, nossa consulta preparada se torna:

```
PreparedStatement consulta = con.prepareStatement(
      "SELECT livros.preco, livros.titulo " +
            "FROM livros, publicadores " +
            "WHERE livros.idpublicador = publicadores.idpublicador " +
                  "AND publicadores.nome = ?");
```

Antes de executar a instrução preparada, você deve ligar as variáveis aos valores reais, com um método set. Assim como nos métodos ResultSet get, existem diferentes métodos set para os diversos tipos. Aqui, queremos configurar uma string como um nome de editora.

```
consulta.setString(1, "Pearson Education Inc.");
```

O primeiro argumento é número da posição que queremos configurar. A posição 1 denota o primeiro "?". O segundo argumento é o valor que queremos atribuir à consulta SQL.

Se você reutilizar uma consulta preparada que já executou e a consulta tiver mais de uma variável, todas as variáveis permanecerão ligadas à medida que você as configura, a não ser que as altere com um método set. Isso significa que você só precisa chamar set nas variáveis que mudam de uma consulta para outra.

Uma vez que todas as variáveis tiverem sido ligadas a valores, você pode executar a consulta:

```
ResultSet rs = consulta.executeQuery();
```

Você processa o conjunto de resultado normalmente. Aqui, incluímos as informações na área de texto result.

```
System.out.println("Preço\t\tPublicador");
System.out.println("-------------\t---------------------------------");
while(rs.next()) {
      System.out.println(rs.getString(1) + "\t\t" + rs.getString(2));
}
con.close();
```

Este código apresentará o resultado abaixo para esta consulta:

```
Preço             Publicador
------------      ---------------------------------
36.55             Effective Java 3rd Edition
47.49             Core Java SE 9 for the Impatient
47.68             Core Java Volume I--Fundamentals
```

Atualizando Preços

O recurso de atualização de preços será implementado como uma instrução UPDATE simples. Assim chamaremos `execute` e não `executeQuery`, pois a instrução UPDATE não retorna um conjunto de resultados e não precisamos de um. O valor de retorno de `execute` é a contagem de linhas alteradas.

A execução da atualização de preço é bastante complexa, pois a cláusula WHERE da instrução UPDATE precisa do código da editora e conhecemos apenas o nome dela. Esse problema á resolvido com uma consulta aninhada.

```
UPDATE livros
      SET preco = preço + valor_da_alteração
      WHERE livros.idpublicador =
            (SELECT idpublicador
                  FROM publicadores
                  WHERE nome = nome_do_publicador)
```

Seguindo da mesma forma como na consulta os argumentos na atualização preparada é indicada com uma “?”. Assim, nossa atualização preparada se torna:

```
PreparedStatement atualiza = con.prepareStatement(
      “UPDATE livros “ +
            “SET preço = preço + ? “ +
            “WHERE livros.idpublicador =“ +
                  “(SELECT idpublicador ” +
                        “FROM publicadores “ +
                        “WHERE nome = ?)”);
```

Note que para a atualização temos dois argumentos à informar.

```
atualiza.setString(1, "Pearson Education Inc.”);
atualiza.setDouble(2, 5.0);
```

A primeira posição recebe o nome do publicador e a segunda posição recebe o valor a ser aplicado ao preço. Agora basta atualizar.

```
atualiza.execute();
```

Conjunto de Resultados Roláveis e Atualizáveis

Os aprimoramentos mais úteis no JDBC estão na classe ResultSet. Conforme você já viu, o método `next` da classe ResultSet faz a interação sobre as linhas de um conjunto de resultados. Normalmente, você quer que o usuário possa mover-se para frente e para trás no conjunto de resultado. Mas o JDBC não tinha um método `previous`. Os programadores que quisessem implementar interação inversa tinha de colocar os dados do conjunto de resultado em cache, manualmente. Os conjunto de resultados roláveis do JDBC permitem que você se mova para frente e para trás em um conjunto de resultados e pule para qualquer posição do conjunto.

Além disso, quando você apresenta o conteúdo de um conjunto de resultados para os usuários, eles podem ficar tentados a edita-lo. Se você fornecer um modo que possa ser editado para seus usuários, terá de certificar-se de que as edições dos usuários sejam enviadas de volta para o banco de dados. No JDBC, você tinha de programar instruções UPDATE. No JDBC, você pode simplesmente atualizar as entradas do conjunto de resultado e o banco de dados á atualizado automaticamente.

Conjuntos de Resultados Roláveis

Para obter conjunto de resultados roláveis a partir de suas consultas, você deve obter um objeto Statement diferente, com o método:

```
PreparedStatement stmt = con.prepareStatement(comando, tipo, concorrência);
```

Os valores possíveis de tipo estão relacionados na Tabela a seguir:

TYPE_FORWARD_ONLY	O conjunto de resultados não é rolável
TYPE_SCROOL_INSENSITIVE	O conjunto de resultados á rolável, mas não é sensível às alterações do banco de dados.
TYPE_SCROOL_SENSITIVE	O conjunto de resultados á rolável e é sensível às alterações do banco de dados.

VALORES DE TIPO DE CONJUNTOS DE RESULTADOS

Os valores possíveis de concorrência estão relacionados na Tabela a seguir:

CONCUR_READ_ONLY	O conjunto de resultados não pode ser usado para atualizar o banco de dados.
CONCUR_UPDATABLE	O conjunto de resultado pode ser usado para atualizar o banco de dados.

VALORES DE CONCORRÊNCIA DE CONJUNTOS DE RESULTADOS

Por exemplo, se você quer simplesmente rolar por um conjunto de resultado, mas não quer quer editar seus dados, então use:

```
PreparedStatement stmt = con.prepareStatement(“SELECT * FROM livros”,
            ResultSet.TYPE_SCROOL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

Todos os conjuntos de resultados que são retornados por chamadas do método abaixo, são roláveis.

```
ResultSet rs = stmt.executeQuery();
```

Um conjunto de resultados rolável tem um cursor que indica a posição corrente.

NOTA: Na verdade, um driver de banco de dados pode não ser capaz de atender seu pedido de cursor rolável ou atualizável. (Os métodos supportsResultSetType e supportsResultSetConcurrency da classe DatabaseMetaData informam quais tipos e modos de concorrência são aceitos por um banco de dados em particular.) Mas, mesmo que um banco de dados ofereça suporte a todos os modos de conjunto de resultados, uma consulta em particular poderia não produzir um conjunto de resultados com todas as propriedades solicitadas. (Por exemplo, o conjunto de resultados de uma consulta complexa talvez não possa ser atualizado.) Nesse caso, o método executeQuery retorna um ResultSet com menos capacidades e inclui um aviso (warning) no objeto conexão. Você pode recuperar avisos com o método getWarnings da classe Connection. Alternativamente, você pode usar os métodos getType e getCurrency da classe ResultSet, para descobrir qual modo um conjunto de resultados realmente possui. Se você não verificar os recursos do conjunto de resultados e executar uma operação são suportada, como previous, em um conjunto de resultados não rolável, será lançada uma SQLException.

A rolagem é muito simples. Você usa o código abaixo para rolar para trás.

```
if(rs.previous()) ...
```

O método retorna true, se o cursor estiver posicionado sobre uma linha real; false, se estiver posicionado antes da primeira linha.

Você pode mover o cursor para trás ou para frente por um número de linhas, com o código:

```
rs.relative(n);
```

Se n for positivo, o cursor se move para frente. Se n for negativo, ele se move para trás. Se n for zero, a chamada não tem efeito. Se você tentar mover o cursor fora do conjunto corrente de linhas, ele será configurado de forma a apontar após a última linha ou antes da primeira, dependendo do sinal de n. Desta forma, o método retorna false e o cursor não se move. O método retorna true, se o cursor estiver sobre uma linha real.

Alternativamente, você pode configurar o cursor para um número de linhas específico:

```
first
last
beforeFirst
afterLast
```

Finalmente, os métodos abaixo testam se o cursor está em uma dessas posições especiais.

```
isFirst
isLast
isBeforeFirst
isAfterLast
```

Usar um cursor rolável é muito simples. O trabalho árduo de colocar os dados da consulta em cache é realizado nos bastidores pelo driver do banco de dados.

Conjuntos de Resultados Atualizáveis

Se você quiser editar os dados do conjunto de resultados e ter as alterações automaticamente refletidas no banco de dados, será necessário criar um conjunto de resultados atualizável. Os conjunto de resultados atualizáveis não precisam ser roláveis, mas se você apresentar dados para um usuário para edição, normalmente vai querer permitir também a rolagem.

Para obter conjuntos de resultados atualizáveis, você cria uma instrução como a seguinte:

```
PreparedStatement stmt = con.prepareStatement("SELECT * FROM livros",
            ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Assim, os conjuntos de resultados retornados por uma chamada a executeQuery serão atualizáveis.

Por exemplo, suponha que você queira aumentar os preços de alguns livros, mas não tenha um critério simples para executar um comando UPDATE. Você pode fazer a iteração por todos os livros e atualizar os preços com base em condições arbitrárias.

```
ResultSet rs = stmt.executeQuery();

while(rs.next()) {
      if(...) {
            double aumento = ...;
            double preço = rs.getDouble(“preco”);
            rs.updateDouble(“preco”, preço + aumento);
            rs.updateRow();
      }
}
```

Existem métodos `updateXxx` para todos os tipos de dados que correspondam aos tipos SQL, como `updateDouble`, `updateString` etc. Assim como nos métodos `getXxx`, você especifica o nome ou o número da coluna e especifica o novo valor para o campo.

O método `updateXxx` altera apenas os valores na linha em si, não o banco de dados. Quando você tiver terminado com as atualizações de campo em uma linha, deve chamar o método `updateRow`. Esse método envia todas as atualizações da linha corrente para o banco de dados. Se você mover o cursor para outra linha, sem chamar `updateRow`, todas as atualizações serão descartadas do conjunto de linhas e elas nunca serão comunicadas para o banco de dados. Você também pode chamar o método `cancelRowUpdades` para cancelar as atualizações na linha corrente.

O exemplo anterior mostra como você modifica uma linha existente. Se você quiser incluir uma nova linha no banco de dados, use primeiro o método `moveToInsertRow`, para mover o cursor para uma posição especial, chamada linha de inserção. Você constrói uma nova linha na posição da fila de inserção, executando instruções `updateXxx`. Finalmente, quando você tiver terminado, chame o método `insertRow` para inserir a nova linha no banco de dados. Quando você tiver terminado de inserir, chame `moveToCurrentRow` para mover o cursor de volta para a posição anterior à chamada de `moveToInsertRow`. Aqui está um exemplo.

```
rs.moveToInsertRow();
rs.updateString(“titulo”, "Effective Java 3rd Edition");
rs.updateString(“isbn”, "978-0134685991");
rs.updateString(“idpublicador”, "01734");
rs.updateString(“site”, "www.pearsoned.com");
rs.updateDouble(“preco”, 36.55);
rs.insertRow();
rs.moveToCurrentRow();
```

Note que você não tem influência sobre onde os dados são inseridos no conjunto de resultados ou no banco de dados.

Finalmente, você pode excluir a linha que está sob o cursor.

```
rs.deleteRow();
```

O método `deleteRow` remove imediatamente a linha do conjunto de resultados e do bando de dados.

Os métodos `updateRow`, `insertRow` e `deleteRow` da classe ResultSet proporcionam a você o mesmo poder que a execução dos comandos SQL UPDATE, INSERT e DELETE. Entretanto, os programadores que estão acostumados com a linguagem de programação Java acharão mais natural manipular o conteúdo do banco de dados através de conjuntos de resultados do que construindo instruções SQL.

Referências

Java 9 Modularity
Sander Mak & Paul Bakker
O'Reilly Media Inc. - 2017

Understanding Java 9 Modules
Paul Deitel
Java Magazine Sep/Oct 2017

The Evolving Nature of Interfaces
Michael Kölling
Java Magazine Sep/Oct 2016

Programando em Java2 - Teoria & Aplicações
Rui Rossi dos Santos
Axcel Books - 2004

Core Java2 - Volume I - Fundamentos
Cay S. Horstmann & Gary Cornell
The Sun Microsystems Press - Série Java - 2003

Java Programming
Nick Clements, Patrice Daux & Gary Williams
Oracle Corporation - 2000

http://www.oracle.com/events/us/en/java8/index.html

http://www.techempower.com/blog/2013/03/26/everything-about-java-8/

http://docs.oracle.com/javase/8/docs/api/java/util/function/package- summary.html

http://docs.oracle.com/javase/tutorial/index.html

http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html