Capítulos de apostila de Java
(última alteração em 1999)

Programação em Java

Helder da Rocha
helder@ibpinet.net

JAVA 1

Introdução Prática

ESTE MÓDULO OFERECE UMA VISÃO GERAL DA LINGUAGEM JAVA, seus recursos e conceitos fundamentais. Apresenta a documentação e mostra como consultá-la. Apresenta as ferramentas do ambiente de desenvolvimento (JDK). No final, são criados pequenos programas em Java que serão compilados e executados.

Índice deste módulo

Objetivos

No final deste módulo você deverá ser capaz de:

- Conhecer as principais características da linguagem Java que a distinguem de outras linguagens.
- Identificar, ao analisar um programa fonte em Java, suas principais estruturas como classes, métodos e declaração de variáveis.
- Saber compilar e executar uma aplicação Java. Saber usar um applet Java (de maneira genérica).
- Saber usar o ambiente de desenvolvimento para editar, salvar, compilar, executar e depurar programas em Java.
- Saber usar a documentação em hipertexto distribuída pela Sun para encontrar classes e métodos.

1.1. O que é Java?

Por que aprender Java? Qual o sentido de se aprender mais uma linguagem de programação? As outras que já existem já não bastam? É verdade que Java só serve para fazer animações na Web? Programas em Java são muito lentos, não são? Afinal, o que é Java?

Talvez por ter surgido numa época em que as informações se espalham rapidamente e de forma caótica, Java tem sido objeto de discussões que ora exageram nas suas qualidades, ora desprezam suas potencialidades, sem contar as vezes que a linguagem é confundida com outras linguagens como C ou JavaScript. O objetivo desta seção é esclarecer essas dúvidas e definir de uma forma clara que é Java, de acordo com o que dizem seus criadores.

O nome Java é usado pela Sun – a empresa que a trouxe ao mundo – para se referir não só à linguagem de programação, mas a toda uma arquitetura que envolve, além da linguagem, um ambiente de execução interpretado (a Máquina Virtual Java ou *Java Virtual Machine* - JVM) e uma plataforma de execução e desenvolvimento (bibliotecas dinâmicas, compiladores, etc.).

Uma das características mais singulares da linguagem e plataforma Java é o fato de poderem ser usadas, no desenvolvimento e na execução, em praticamente qualquer plataforma. Seu código-fonte, pode ser compilado em qualquer plataforma que possua um ambiente de desenvolvimento (*Java Development Kit* – JDK). O código é compilado para uma linguagem de máquina *virtual* que roda sem modificações em um amplo universo de plataformas e sistemas operacionais

diferentes (basta ter uma plataforma Java com o JVM instalado). Essa característica é fundamental para o sucesso de Java no mundo das aplicações distribuídas.

Mas antes que falemos mais de Java, vamos antes conhecer melhor suas características. Iniciaremos com um pouco de história e depois deixaremos que a Sun descreva os adjetivos que usa para definir a linguagem.

A história

Java nasceu das cinzas de um projeto fracassado. Inicialmente foi uma linguagem desenvolvida para programar chips de aplicativos de consumo, como PDAs, fornos de microondas, etc. A equipe que desenvolvia o projeto na Sun Microsystems, liderado por James Gosling, tinha inicialmente pensado em utilizar C ou C++, mas descobriu logo cedo que a linguagem não se adequava ao projeto. Como resultado, em 1990, Gosling começou a projetar uma nova linguagem que ele batizou de Oak (segundo ele, por causa de um carvalho – *oak*, em inglês – que ele via da sua janela). O nome posteriormente teve que ser mudado por que já existia uma outra linguagem registrada com o mesmo nome. A lenda diz que a idéia do nome veio numa visita à lanchonete que servia café, que os americanos costumam batizar com os mais variados nomes. Na época, o apelido era *Java*, referindo-se à ilha de origem dos grãos (na verdade, na hora do consumo, pouco importava se o café era colombiano, brasileiro ou asiático – era Java).

O projeto avançou, mas o mercado decepcionou e a Sun decidiu alterar seu rumo, desta vez investindo no ramo da TV interativa. O programa durou dois anos e consumiu milhões de dólares. Consistia no desenvolvimento de um protótipo de controle para TV interativa, usando uma linguagem desenvolvida especialmente para este fim (Java). Foi gasto uma fortuna. Criaram até mascote (o *Duke*... um bicho que parece com um dente de cabeça para baixo e que agora é mascote do Java). Quando o projeto finalmente foi apresentado em 1993, descobriu-se que o mercado de TV interativa não existia, e não havia uma previsão para quando iria deslanchar.

O financiamento para o projeto estava para ser cortado e a equipe dissolvida e transferida para outros projetos quando a Sun decidiu mudar drasticamente o objetivo do projeto, e focar o seu desenvolvimento na Web –

uma novidade que tinha acabado de surgir e já começava a fazer sucesso. A partir daí, a equipe trabalhou em um ritmo alucinante e em segredo até liberar a primeira versão alfa da linguagem, em maio de 1995.

As primeiras notícias ouvidas de Java eram sobre uma tecnologia que possibilitava colocar animações em uma página Web (na época, ainda não existiam as imagens GIF animadas). Isto porém só era possível com um browser chamado *HotJava*. O browser permitia que um programa fizesse parte de uma página, assim como uma imagem é integrada nas páginas Web. O programa era transferido pela rede e rodava na máquina do usuário, aparecendo dentro de uma parte da tela gráfica do *HotJava*. Esses programas, que rodavam dentro de páginas Web, foram chamados de *Applets*. As pessoas começaram a ver que o mundo estava cheio de applets, quando o *Netscape Navigator* – browser usado por 86% da população da Web na época – passou a suportá-los também.

Muitos ainda achavam que Java era só mais um brinquedo para tornar a Web mais atraente. Mas logo surgiram alternativas mais simples e leves para realizar animações e outras coisas antes só possíveis com Java. O interesse pela linguagem, porém, não morreu, pois a estratégia já havia funcionado: as pessoas começavam a descobrir que Java era mais que uma mera tecnologia para animar a Web. Era uma poderosa linguagem que podia ser usada para desenvolver aplicações, como as que se faz hoje usando C++, Delphi ou VB. Tinha, ainda, uma série de vantagens que as outras não possuíam: era orientada à rede, era mais segura, tinha recursos que a tornavam mais robusta e, apesar disso ainda conseguia ser menos complicada que várias outras linguagens com bem menos recursos. Tinha também o que era mais importante para um mundo onde a plataforma deixava o desktop e passava a ocupar as redes mundiais: era independente de plataforma. Com Java, finalmente era possível desenvolver um único programa, compilá-lo uma vez e rodá-lo em quase qualquer lugar. E esse tornou-se o lema da linguagem: "Escreva uma vez. Rode em qualquer lugar."

Java segundo seus criadores

Segundo a Sun, Java pode ser definida com uma série de adjetivos. Java é:

- uma linguagem *simples* e *familiar*;
- *orientada a objetos*;
- *compilada* e *independente de plataforma*;

- com *suporte à programação concorrente*;
- com *coleta de lixo*;
- *robusta*, com '*tipagem' forte*,
- *segura*,
- *extensível* e *estruturada*.

Vejamos alguns desses conceitos em detalhe:

Simples e familiar

Java é muito parecida com C – a linguagem mais popular do mundo para o desenvolvimento de software básico. Se você já conhece C ou C++, irá dominá-la em pouco tempo. Esta característica é provavelmente uma das principais razões da popularidade da linguagem. Várias outras linguagens bem melhores que C++ foram inventadas nos últimos anos, mas não atraíram a mesma atenção pois exigiam que o programador aprendesse uma sintaxe completamente nova, o que exigia dedicação e paciência – recursos caros para a maioria dos programadores.

Apesar da semelhança, Java não herdou as complexidades do C e é muito mais simples que C++. Todos os recursos do C/C++ que podiam ser substituídos com construções mais simples (como *estruturas*, *unions* e *typedef*) foram deixados de lado. Diferente de C, Java não tem ponteiros, arquivos de cabeçalho, pré-processadores, estruturas, uniões, matrizes multidimensionais, gabaritos e nem suporta sobrecarga de operadores.

Orientada a Objetos

Java é uma verdadeira linguagem *orientada a objetos*. Tudo em Java é objeto. A única exceção são os tipos simples como números e variáveis booleanas, que não são objetos por questões de desempenho. Mesmo assim, podem ser encapsulados em objetos quando necessário.

A orientação a objetos permite que programas sejam decompostos em estruturas de dados menores, com variáveis locais próprias e tarefas (operações) bem definidas. Um programa orientado a objetos consiste de várias dessas estruturas (os *objetos*) interligados que solicitam tarefas (através de *métodos*) entre si. Os programas podem ficar bem mais simples que os desenvolvidos usando decomposição orientada a procedimentos (procedural), pois um programa maior pode combinar vários objetos menores (e talvez bastante complexos) sem que

precise saber nada a não ser como usar seus métodos (que em geral são poucas e bem definidas).

Os arquivos em Java são organizados em *classes*, que são usadas para produzir objetos. As classes também podem herdar características (métodos e variáveis) de outras classes.

Java não suporta herança de múltiplas classes (característica presente em algumas linguagens orientadas a objeto como C++) já que o seu uso pode aumentar a complexidade sem trazer benefícios correspondentes. Mas existem casos onde a herança múltipla é benéfica. Para estes casos Java tem uma solução que permite o desenvolvimento de código com alto grau de reuso: as *interfaces*. Um programa construído usando interfaces pode ter módulos acrescentados no futuro sem que o seu código existente precise ser alterado (ou sequer compilado).

Em Java não existem variáveis globais ou funções independentes. Toda variável ou método pertence a uma classe ou objeto e só pode ser utilizada através dessa classe ou objeto.

Compilada e Independente de Plataforma

Um programa escrito em Java precisa ser compilado antes de ser executado. O compilador traduz o código-fonte e gera um código em *linguagem de máquina* para um *microprocessador virtual*. Esses arquivos são chamados *arquivos de classe* e sua linguagem de máquina é freqüentemente chamada de *bytecode*. Cada programa Java consiste da implementação de, no mínimo, *uma* classe. Depois de compilado, o programa-objeto pode ser executado em qualquer plataforma onde exista uma Máquina Virtual Java (JVM) instalada.

A compilação de um programa em C++ realiza a tradução do código-fonte da linguagem em instruções que são interpretadas pelo microprocessador da máquina onde foi compilado. O programa então só roda em outra máquina que tenha o mesmo tipo de processador. Já um programa em Java, quando compilado, gera instruções (os *bytecodes*) para o microprocessador da máquina virtual. Existem implementações desse microprocessador virtual para várias plataformas e o programa então rodará em qualquer uma delas.

O código Java consegue ser independente de plataforma porque a compilação é feita para uma máquina imaginária. Para que uma plataforma seja

capaz de rodar programas em Java, ela deve ter uma implementação que permita a emulação da máquina virtual.

Por ter suas instruções interpretadas por um software (processador virtual), os programas em Java são mais lentos que os escritos em C ou C++ (portanto, não é a *linguagem* que é lenta, mas a *plataforma*). Há, porém, vários meios de melhorar esse desempenho. Os interpretadores Java com *compiladores Just-In-Time* (JIT) já são praticamente onipresentes entre as plataformas Java mais populares (as primeiras não eram JIT). Esse tipo de sistema, converte as instruções em *bytecodes* para instruções do microprocessador na hora da execução, fazendo com que programas escritos em Java não percam em desempenho para programas escritos em C ou C++. Em alguns casos, porém, a compilação pode demorar e talvez não seja desnecessária a não ser naqueles trechos de código que exijam mais do processador. Para solucionar esse problema há uma nova geração de interpretadores-compiladores que se adaptam a essas condições. A Sun os chama de *HotSpot*. Já existem várias situações onde programas em Java rodam tão rápidos ou até mais rápidos que programas escritos em C++.

Robusta

Java é uma linguagem que tem *tipagem* de dados *forte*: ela exige que o *tipo* de objetos e números seja explicitamente definido durante a compilação. O compilador não deixa passar qualquer indefinição em relação ao tipo de dados. Esta característica garante uma maior segurança do código e o torna menos sujeito a erros, além de facilitar a depuração.

Programas em Java não provocam (sozinhos) core-dumps ou GPFs (*General Protection Fault*). Enquanto programas escritos em C ou C++ podem alterar qualquer posição da memória do computador, os programas escritos na linguagem Java não têm acesso direto à memória e deixam o controle a cargo do sistema operacional.

O sistema de gerenciamento de memória de Java libera os programadores da tarefa de se preocupar em liberar memória usada (veja tópico seguinte). Um processo chamado de coletor de lixo (*garbage collector*) opera sempre em uma rotina (*thread*) de baixa prioridade, liberando recursos que não são mais utilizados automaticamente.

E quando ocorrem erros ou situações inesperadas em tempo de execução, Java possui um meio de lidar com elas e se recuperar do erro se possível. O controle de exceções (condições excepcionais) é uma parte fundamental da linguagem e em muitos casos seu uso é obrigatório.

Todos esses recursos, tornam os programas escritos em Java mais robustos, pois torna os erros menos freqüentes tanto na fase de desenvolvimento como na fase de execução. O resultado é uma maior produtividade no desenvolvimento, reduzindo o custo para se ter programas com menos *bugs* e mais confiáveis.

Com coleta de lixo

A coleta de lixo é uma técnica automática de liberação de memória. Muitas linguagens permitem que ao programador alocar memória em tempo de execução. Esta alocação consiste geralmente no retorno de um ponteiro que indica o início do bloco de memória que foi alocado. Quando os dados armazenados naquela posição de memória não são mais necessários, o programa deve liberar os recursos para que ela possa ser reutilizada e evitar que o sistema pare por falta de memória.

Nas linguagens de programação comuns (como C e C++), é o programador que deve manter um controle da memória que foi alocada e liberá-la quando ela não mais for utilizada. Isto é uma tarefa complexa que exige disciplina e extrema atenção do programador. Geralmente, o programador que escreve o código esquece de liberar memória (ou libera quando ainda não pode liberar) e o problema só é descoberto quando o software já está em versão beta. Nesse estágio, uma dúzia de programadores vão passar noites tentando corrigir os bugs e o cronograma de entrega do software vai mais uma vez ser adiado, e o custo vai subir.

O sistema de coleta de lixo tira esta responsabilidade do programador, aumentando a produtividade, reduzindo os custos. Através de uma linha de execução (*thread*) de baixa prioridade, o sistema de coleta de lixo mantém um controle da memória alocada e conta o número de referências que existem para cada ponteiro de memória. Nos intervalos em que a JVM está inativo, o coletor de lixo (*Garbage Collector* – GC) verifica quais os ponteiros de memória que não têm mais referências apontando para eles marca a o endereço de memória correspondente como livre.

O GC é acionado automaticamente pelo sistema e o programador não tem acesso a ele. Ele pode, porém, influenciar o GC reinicializando as referências ou chamando o GC diretamente (através de função), mas a operação só ocorre quando a JVM permitir. Não há como forçar a coleta de lixo.

Dinâmica (extensível)

A linguagem Java foi projetada para se adaptar a um ambiente dinâmico, em constante evolução. Um programa pode, por exemplo, receber código binário (*bytecode*) pela Internet, carregá-lo e se estender automaticamente com novas classes (módulos). Possui uma representação (sintaxe) de tempo de execução que permite que o programa saiba a que classe pertence um objeto, na hora em que o recebe. Isto permite a inclusão dinâmica de classes que podem estar em qualquer lugar da rede.

Java também suporta a integração com métodos (funções) nativos de outras linguagens. Dessa forma, pode-se desenvolver aplicativos híbridos em Java e C ou C++, aproveitando o grande volume de código existente hoje nessas linguagens. Usando a linguagem JavaIDL (parte do JDK) e CORBA (tecnologia de integração orientada a objetos), uma aplicação Java pode se comunicar com qualquer outro programa orientado a objetos através da rede.

Segura

Por ter uma *tipagem* de dados forte, somente permitir acesso a campos pelo nome (e não por endereço), não ter aritmética de ponteiros nem qualquer tipo de acesso direto à posições de memória, um programa compilado em Java pode ser verificado antes de ser executado. A verificação dos *bytecodes* é realizada nos browsers Web que suportam Java para garantir que os *applets*, transferidos pela Internet, não estejam violando as restrições da linguagem e não possam provocar danos no computador do usuário. Depois da verificação, os applets podem ser otimizados pelo sistema de execução, para garantir um melhor desempenho.

A linguagem também fornece um esquema de segurança baseado em certificados. Com eles, pode-se classificar certos applets como "confiáveis" com base em um certificado garantido pelo seu fabricante e ampliar as suas capacidades (applets normalmente têm várias restrições de segurança). Com esse artifício, será possível desenvolver applets mais úteis com capacidade de escrever em disco, imprimir, acessar mais de um endereço de rede, etc.

Como Java não dá acesso direto à memória, torna-se muito difícil o desenvolvimento de vírus, da forma como são feitos hoje, usando a linguagem Java. Pode haver, porém, programas hostis criados para causar transtornos aos usuários. A solução contra este tipo de programa pode estar nos certificados de segurança.

A segurança é garantida por uma classe Java chamada *Security Manager*. Nos browsers, o usuário não pode alterá-la, mas nas aplicações distribuídas desenvolvidas pelo programador, diferentes níveis de segurança poderão ser definidos. Nesses casos, a segurança é controlada exclusivamente pelo programador. Mas nos applets, não. Ao receber um applet em seu browser, a máquina virtual nele instalada sempre observa as três etapas abaixo:

- O carregamento do código
- A verificação do código
- A execução do código

A segunda etapa, de verificação, só ocorre com applets que vêm pela rede. Nas aplicações ou applets locais, apenas a primeira etapa e a última são realizadas. O *Class Loader* – carregador de classes sabe se a classe é local ou remota na hora em que recebe o programa Java (do disco local ou de um endereço remoto). Programas locais então são considerados confiáveis e têm menos restrições. Programas remotos têm que passar pelo *Bytecode Verifier* – o verificador de código, que irá dizer se o código foi corrompido ou não.

Suporte à programação concorrente (multithreading)

Programas em Java podem ter mais de uma linha de execução ocorrendo ao mesmo tempo. Os programadores podem definir quando e com que prioridade certas linhas de execução serão rodadas. O *multithreading* é essencial em aplicações gráficas. Sem ele não seria possível que os programas realizassem outras tarefas enquanto o usuário interagisse com ele. A programação de *threads* é trabalhosa em qualquer linguagem, inclusive em Java, mas Java a torna bem menos complexa ao permitir que ela seja realizada com classes e objetos da linguagem e não exigir que o programador faça chamadas ao sistema operacional (como ocorre com C ou C++). O resultado é que programadores são estimulados a usar threads sempre que o seu uso trouxer benefícios ao programa. Embora o controle e

agendamento de *threads* dependam do sistema operacional, é possível desenvolver programas *multithreaded* independentes de plataforma em Java.

Ambientes de execução

Java pode ser empregada em vários ambientes de execução. Os mais conhecidos são o sistema operacional (programas independentes), o browser (*applets*) e o servidor (*servlets*). Mas as plataformas que suportam Java não se limitam a *mainframes* e computadores desktop. Java pode ser usada em telefones celulares, fornos de microondas, controles remotos de TV, automóveis e até em anéis, colares, braceletes e cartões magnéticos! O melhor de tudo é que os programas desenvolvidos para essas plataformas podem ser criados e prototipados em computadores comuns, reduzindo imensamente o custo de desenvolvimento. Você pode, por exemplo, desenvolver software para cartões magnéticos (*smartcards*) baixando o *JavaCard* API do site da Sun e usando-o para compilar seus programas. Pode desenvolver software para telefones celulares usando a API *PersonalJava*. Vários aparelhos hoje em dia possuem máquinas virtuais Java embutidas e são clientes em potencial dos softwares que você poderá desenvolver com Java.

O mais recente avanço na tecnologia de plataformas Java é o *Jini* – uma arquitetura que permite o desenvolvimento de aplicações distribuídas entre quaisquer tipo de aparelho. Tem como objetivo fazer valer a máxima *Write once, run anywhere* (escreva uma vez, rode em qualquer lugar) e deverá ampliar de forma significativa o mercado para os desenvolvedores Java.

O que promete a linguagem Java?

Em uma rede heterogênea, que consiste de milhões de máquinas diferentes ligadas entre si, as características da linguagem Java a colocam em posição confortável como a linguagem ideal para o desenvolvimento de aplicações para a Internet e sistemas distribuídos.

Os *bytecodes* compilados são transferidos através da rede e rodam em qualquer máquina ligada à Internet, seja como uma applet, através de um browser, seja como uma aplicação independente. Se o seu aplicativo favorito foi escrito em Java, não importa se você tem uma máquina Solaris, um PC rodando Windows, um servidor NT ou um Macintosh. Ele irá rodar de forma quase igual em qualquer dessas máquinas.

Java deverá causar uma redução do custo de aplicativos que precisam rodar em várias plataformas. Com linguagens tradicionais, era necessário portar e compilar o código-fonte em cada plataforma diferente, e tratar os *bugs* que apareciam em cada uma delas individualmente. Depois, ainda ter documentação e suporte para cada plataforma, e embalagens diferentes para cada versão. Se o programa for escrito em Java, só haverá uma compilação, uma documentação e um produto comercial que poderá ser adquirido por qualquer usuário, independente de sua plataforma de trabalho.

A forma como se usa software também poderá mudar. Com uma rede suficientemente rápida, um usuário que precisar ocasionalmente usar um aplicativo qualquer poderia alugá-lo em vez de comprar todo um pacote de software. O programa seria transferido para a sua máquina, ele realizaria o seu trabalho e depois o descartaria, pagando apenas uma taxa pela utilização. Esta é a filosofia do *Network Computer* (NC) e dos provedores de aplicações.

Todas essas idéias são possíveis com Java. Em 1997 elas pareciam utópicas, irreais. A linguagem também só tinha dois anos e apresentava problemas de compatibilidade, a rede era lenta, entre outros problemas. Hoje a linguagem está bem mais estável com a plataforma Java 2 (JDK1.2) e muitas das utopias do passado já são realidade. Agora chegou o Jini com a promessa de colocar Java em todos os aparelhos do lar. E então, por que aprender Java? Será que vale a pena? Se você acha que sim, então, mãos à obra!

1.2. Introdução Prática

Nesta seção nós iremos introduzir os conceitos básicos de Java, como sua estrutura léxica, tipos de dados, expressões e outras características comuns à qualquer linguagem de programação. Como é impossível estudar Java sem levar em conta a sua natureza orientada a objetos (todo programa Java começa com uma declaração de classe), será inevitável depararmos com exemplos de criação de objetos, utilização de métodos, etc. Estes tópicos serão explicados aqui de forma superficial, mas merecerão uma abordagem mais detalhada em capítulos posteriores.

O Java Development Kit

A melhor forma de introduzir os conceitos de uma linguagem é com exemplos reais. É o que faremos nesta seção. Você pode testar todos os programas usados nos exemplos no seu próprio computador. Para isto é preciso que tenha no mínimo uma das configurações seguintes:

- O Java Delopment Kit (JDK 1.1 ou superior) e um editor de textos (Notepad, Vi, WinEdit, etc.); ou
- Um Ambiente Integrado de Desenvolvimento (*Integrated Development Environment* – IDE) para a linguagem Java como o Microsoft Visual J++, Sun Java Studio, Symantec Visual Café, Borland JBuilder, Oracle JDeveloper ou outro.

Um IDE é ideal para desenvolver aplicações em Java com *produtividade*. Possui diversas ferramentas gráficas para gerenciar projetos, gerar código, depurar programas e reutilizar módulos dinâmicos. Um IDE, porém, não é o ambiente ideal para *aprender* Java. Usar o Borland JBuilder ou o Oracle JDeveloper, por exemplo, é fortemente recomendado no dia-a-dia do desenvolvimento de software em Java, mas é extremamente importante que você saiba Java e se familiarize com a linguagem antes! Se você sabe Java, não ficará dependente de uma ou outra ferramenta. Você terá condições de ir além da ferramenta quando e se for preciso.

Como o JDK é gratuito, e pode ser descarregado pela Internet, as instruções de compilação e execução mostradas aqui serão referentes a ele. Se você pretende acompanhar os exemplos deste capítulo, instale e configure o JDK agora (veja se ele já não está configurado na sua máquina). O funcionamento do JDK é dependente de plataforma. Leia o README e instruções para saber como usá-lo em sua máquina. As instruções deste capítulo supõem que o seu JDK irá rodar em um PC ou máquina Unix (se você estiver usando um Mac, consulte a documentação para saber como executar, compilar e editar arquivos).

O JDK 1.1 (para Windows e Solaris) consiste do seguinte:

- *Java API*: todas as classes e interfaces, organizadas em *pacotes*;
- Os *códigos-fonte* das classes, interfaces e métodos da API;
- *Applets e aplicações de demonstração* com exemplos de utilização;
- *Máquina Virtual Java* (java):
- *Visualizador de Applets* (appletviewer);

- *Ferramentas de desenvolvimento:* compilador Java (javac), gerador de métodos nativos C (javah), gerador de documentação (javadoc), debugger (jdb), disassembler (javap), profiler (javaprof), jar e outras.

A documentação do JDK

Além das ferramentas básicas, ainda usaremos a *documentação* em hipertexto para termos acesso a uma referência completa da linguagem Java. Essa referência está em HTML e pode ser consultada através de um browser. A documentação não faz parte do pacote distribuído pela Sun. Precisa ser baixada à parte e tem quase o mesmo tamanho em megabytes que o próprio JDK.

Descarregue a documentação e expanda o arquivo ZIP na raiz do disco onde o seu JDK estiver instalado. A descompressão automaticamente criará um subdiretório /docs/ dentro daquele diretório. Para ler a documentação, abra o arquivo index.html desse diretório no seu browser. Uma referência em hipertexto está disponível na opção "Java Platform API". Durante todo este curso, usaremos esta referência como guia para o desenvolvimento dos programas.

Como conseguir o JDK

Caso você deseje obter uma distribuição do JDK para uso próprio ou em outra plataforma que aquela do laboratório, poderá obtê-la pela Internet. Elas são gratuitas. O JDK1.2 (ou Java 2 platform) está disponível na Sun para os sistemas operacionais Solaris (SunOS), Windows e Macintosh. As versões para outras plataformas são distribuídas por outras empresas cujo endereço se obtém através das páginas da Sun.

Para descarregar a última versão do JDK para Solaris (SPARC ou x86), Windows ou Macintosh, visite a sua home page na JavaSoft em:

```
http://java.sun.com
```

O JDK também existe para outras plataformas. Informações sobre esses produtos podem ser obtidas no mesmo site acima. Algumas versões são:

- OS/2 e AIX:

  ```
  http://ncc.hursley.ibm.com/javainfo/
  ```

- Linux:

  ```
  http://www.blackdown.org
  ```


- NeXT:

  `http://www.next.com`

- HP/UX, AT&T Unix e DEC Alpha OSF/1:

  `http://www.gr.osf.org:8001/projects/web/java/`

Como Preparar e Usar o Ambiente Java da Sun

Depois que você tiver transferido o JDK para o seu disco, instale-o de acordo com as instruções para seu sistema (as instruções para *Windows*, *Solaris* e *Macintosh* estão disponíveis na home page do JDK).

O ambiente utiliza uma propriedade do sistema chamada `CLASSPATH` para informar a localização das bibliotecas de classes durante a compilação ou execução. Em geral, ela é configurada automaticamente, mas, se você estiver utilizando pacotes de terceiros, poderá ser necessário acrescentar o caminho dessas classes ao `CLASSPATH`. Você pode fazer isto a cada compilação ou execução, ou então definir esse caminho no seu sistema. No *Windows95*, você poderá acrescentar no arquivo **`AUTOEXEC.BAT`**, uma declaração do tipo:

```
CLASSPATH = %CLASSPATH%;c:\classes\adicionais\arquivo.jar;.
```

O `CLASSPATH` original, que você não deve sobrepor, contém os caminhos para as bibliotecas fundamentais, essenciais para a compilação e execução dos programas em Java. Bibliotecas são, em geral, distribuídas em arquivos `*.jar` ou `*.zip` que contém as classes organizadas em sistemas de diretórios.

No *Windows95*, você também deve incluir na sua variável `PATH`, o caminho `jdk1.2\bin\`, para que as ferramentas e aplicações do JDK possam ser executadas de qualquer lugar através do *MS-DOS Prompt*. No *Windows*, é interessante que você use o aplicativo `DOSKEY`, que guarda um histórico das linhas de texto digitadas, principalmente se você estiver usando o JDK apenas como ambiente de execução e compilação. Um `AUTOEXEC.BAT` típico para usar o JDK via linha de comando seria:

```
PATH=%PATH%;c:\jdk1.2\bin;
DOSKEY
```

Supondo que o seu ambiente Java foi instalado no drive C:

Depois de instalado o ambiente Java, você poderá rodar e compilar aplicações e applets, usando instruções da linha de comando. Por exemplo, para compilar um arquivo **`MeuProg.java`**, use o **`javac`** da seguinte maneira:

```
javac MeuProg.java
```

O compilador poderá responder com uma série de mensagens de erro ou gerar um ou mais arquivos objeto (`*.class`) com sucesso. Para executar o arquivo compilado **`MeuProg.class`** (se ele for um programa executável) use o programa interpretador **`java`**, omitindo (sempre) a extensão do arquivo:

```
java MeuProg
```

Isto vale até para aplicações gráficas (elas irão restaurar o ambiente gráfico do sistema para executar). Se seu arquivo de classe não for uma aplicação mas for uma applet, você só poderá executá-lo através de uma página HTML, onde ele deve ter sido embutido (adiante, veremos como fazer isto). Ele poderá ser visualizado em um browser ou com o **`appletviewer`** – ferramenta do JDK para testar applets. O argumento do appletviewer deve ser o nome de uma página HTML:

```
appletviewer TesteApplet.html
```

O **`appletviewer`** mostra apenas a applet, ignorando o restante da página HTML. Para ver toda a página com a applet incluída, você precisará de um Browser Web que suporte Java, como o Netscape Navigator ou o Internet Explorer com suporte Java.

Existem ferramentas de desenvolvimento para criar rotinas de instalação e execução profissionais para programas em Java. Elas são comerciais. As mais populares são o *InstallShield* e o *InstallAnywhere*. Elas instalam, na máquina do cliente, o JRE – *Java Runtime Environment* (parte do JDK sem as ferramentas de desenvolvimento) se o usuário já não tiver o ambiente na sua máquina e criam ícones da área de trabalho ou barra de ferramentas do sistema operacional do usuário. O JRE está incluído no JDK e você pode distribuir suas aplicações com ele livremente. É possível também, baixar o JRE da Sun sem o JDK.

Ambiente de desenvolvimento do laboratório

Neste curso, utilizaremos o JDK da Sun com um editor de código shareware chamado *WinEdit* (ambiente PC), que oferece uma interface tipo IDE

ao JDK. O *WinEdit* tem a vantagem de ser configurável, adequado à edição de código por converter tabulações em espaços além de outras vantagens. Ele permite que o usuário configure seus menus ainda para rodar o compilador, o interpretador e o *debugger* Java (JDB). Na versão que iremos utilizar, as palavras reservadas da linguagem Java aparecerão em azul se o arquivo tiver a extensão .java. Então, antes de escrever qualquer coisa em um arquivo do WinEdit, salve-o primeiro com a extensão `.java`. O *Kawa* é bem mais versátil e tem bem mais recursos. Escolha uma das duas ferramentas e mãos à obra.

Exemplos de Programas em Java

A seguir, mostraremos vários programas (didáticos[1]) em Java. Se você quiser digitá-los e testá-los em seu computador (altamente recomendável), tenha o cuidado de digitar o código *exatamente* da forma como é apresentado. Letras minúsculas e maiúsculas *são diferentes* em Java. Se na listagem aparece "String" e você digita "string", com "s" minúsculo, o compilador acusará um erro e o seu programa não será compilado.

Esta regra também vale para o *nome do arquivo*. Mesmo que o *Windows95* não faça distinção entre maiúsculas e minúsculas, quem efetivamente executa o programa compilado é o interpretador Java (seja via browser ou linha de comando) que considera maiúsculas diferentes de minúsculas. Então, se você salvar o arquivo como `Hello.java` e tentar compilar `hello.java`, o compilador acusará um erro, pois não será capaz de encontrar o arquivo. Use sempre os nomes do Windows e nunca os nomes do DOS (`HELLO~1.JAV` não funciona!).

HelloWorld Console

O seguinte programa é o clássico "Hello World" escrito como uma aplicação Java. Depois de compilado e executado ele imprime na tela (linha de comando) as palavras "Bom dia, javaneses!!!!" dez vezes.

```
import java.lang.Object;
import java.lang.String;
import java.lang.System;
```

[1] Programas didáticos geralmente são inúteis e programas muito úteis geralmente não são bons exemplos didáticos. Neste curso mostraremos programas mais úteis logo que tenhamos um pouco mais de experiência com a sintaxe de Java.


```
/* Programa Hello World */

class HelloWorld extends Object {

    public static void main(String[] args) {
        for(int x = 0; x < 10; ++x) {
            System.out.println("Bom dia, javaneses!!!");
        }
    }

}
```

Digite, compile e rode o programa acima no seu computador (os detalhes dependem do ambiente de desenvolvimento que você estiver usando). Se você estiver usando o JDK, faça o seguinte:

1) Crie um diretório para armazenar seus programas Java e salve o programa acima em um arquivo "`hello.java`", nesse diretório;
2) Abra uma janela de comando (MS-DOS Prompt) ou shell e mude para o diretório onde está o programa.
3) Para compilar, rode o compilador Java (`javac`) seguido do nome do seu programa:

   ```
   javac hello.java
   ```

   Se não houver erros, o programa foi compilado e o prompt C:> reaparecerá (se houver erros, volte e corrija-os). Se você listar o conteúdo do seu diretório agora, verá que o compilador criou um arquivo chamado "`HelloWorld.class`". Observe que este é o nome que demos para a classe no programa acima.
4) Finalmente, para rodar o seu programa, chame o interpretador Java seguido do nome da classe (sem a extensão):

   ```
   java HelloWorld
   ```

   O programa, então, deve exibir na tela o texto seguinte:

```
Bom dia, javaneses!!!
Bom dia, javaneses!!!
Bom dia, javaneses!!!
...
```

Como funciona?

O programa consiste de uma declaração de classe (que representa o programa-objeto) e de um método (função) especial (`main()`) que contém as

instruções da principal linha de execução do programa (onde ele começa). Você pode compilar código Java sem o `main()` mas ele não será um programa executável do S.O. É no `main()` que começa a execução da aplicação. Por `main()`, nos referimos na verdade ao bloco:

```
public static void main(String[] args) { ... }
```

Que deve ser digitado EXATAMENTE como acima, com exceção da palavra `args` que é nome de variável e pode ser substituída por qualquer outra (`x`, por exemplo). A *classe* é todo o programa:

```
class HelloWorld extends Object { ... }
```

Isto diz que `HelloWorld` é uma classe Java e que estende a classe `Object`. Todos os nomes em negrito são classes. Em Java, é uma convenção escrever nomes de classes com a primeira letra maiúscula (isto é regra na API). `Object` é uma classe (estrutura ou tipo de dados) que contém uma infraestrutura mínima para que outras classes possam existir. Qualquer classe Java, seja ela da biblioteca fundamental (API), seja ela criada por você, deriva e herda todas as características de `Object`. A extensão (`extends`) caracteriza a herança. Você pode estender `Object` ou outra classe (que descende de `Object`). Nós sempre construímos em cima de uma infraestrutura existente e assim fica mais fácil de escrever programas.

Mas onde está `Object`? Como é que o sistema encontra essa classe? A resposta a essas perguntas está nas três primeiras linhas do programa (as instruções `import`) e no `CLASSPATH`. Quando você instala um programa Java, o JDK, um browser ou qualquer aplicação que utilize a máquina virtual Java, essa aplicação define um `CLASSPATH` que informa ao ambiente de execução onde estão as classes da biblioteca fundamental. Esse `CLASSPATH` é definido automaticamente quando você instala o JDK. Pode ser algo como:

```
C:\jdk1.2\jre\lib\rt.jar
```

O arquivo `rt.jar` contém um sistema de arquivos e diretórios com as classes Java. Ele tem a seguinte estrutura de diretórios:

```
java\ ____ lang\
      |__ awt\ _____ event\
      |          |___ image\
      |       (...)
      |__ applet\
```



```
|  (...)
|__ sql\
```

Cada diretório é chamado de pacote. Dentro dos pacotes há várias classes, por exemplo, dentro de `java\lang\` existem, entre outras, as classes `Object.class`, `String.class` e `System.class` que importamos no início do nosso programa. Podemos concluir, então, que o separador "\" ou "/" dos diretórios é substituído por um ponto ".", quando lidamos com pacotes.

Dentro do pacote java.lang estão classes essenciais em qualquer programa Java. Por causa disto, mesmo que você não importe as classes do java.lang explícitamente, elas são importadas automaticamente pelo sistema. Toda classe também tem que herdar pelo menos de Object, portanto, se nós não tivéssemos declarado isto na cláusula extends, o sistema iria fazê-lo automaticamente. Em outras palavras, o programa funcionaria da mesma forma se tivéssemos programado apenas as linhas abaixo:

```
class HelloWorld {
    public static void main(String[] args) {
        for(int x = 0; x < 10; ++x) {
            System.out.println("Bom dia, javaneses!!!");
        }
    }
}
```

Só há um método (função) no nosso programa. É o `main()`. Neste programa, há no `main()`, uma estrutura de repetição `for`:

```
for(int x = 0; x < 10; ++x) {
   System.out.println("Bom dia, javaneses!!!");
}
```

O texto em sublinhado corresponde a palavras reservadas. A estrutura `for`, que vem seguida de argumentos entre parênteses, abre um bloco entre chaves que só contém uma instrução, que é repetida dez vezes. No próximo capítulo analisaremos em detalhes a sintaxe do `for` que *inicializa*, *testa* e *incrementa* uma variável enquanto é chamada. A instrução que é repetida no `for` é:

```
System.out.println("Bom dia, javaneses!!!");
```

Não vamos entrar em detalhes sobre a sintaxe desta instrução. No momento só precisamos saber que essa instrução imprime o conteúdo de uma cadeia de caracteres (entre aspas) que é passada entre parênteses. Outro detalhe

importante é o ponto-e-vírgula (`;`). Assim como várias outras linguagens de programação, em Java, toda instrução simples deve terminar em ponto-e-vírgula.

A palavra reservada "class" precede o nome da classe que chamamos de `HelloWorld` e todo o código do programa fica dentro desta classe, delimitado pelas chaves `{` e `}`. Dentro da classe, está o método (main) que também envolve com chaves toda a sua implementação. Todo programa em Java tem esta estrutura. Procedimentos (instruções, controle de fluxo, etc.) só podem ocorrer dentro de métodos. Métodos só podem ocorrer dentro de classes. Fora de uma classe só se admite *comentários*, zero ou mais declarações *import* (antes da classe), zero ou uma declaração *package* (antes dos *import*) e outras definições de classe (pode haver mais de uma por programa-fonte).

O nome do método é apenas `main`, mas a sua declaração traz outras palavras: `public`, `static` e `void`. São chamadas de *modificadores* do método. Main é um método especial e deve estar presente em toda rotina *executável* através do S.O. Quando o interpretador Java roda o programa, o primeiro método que ele chama é o `HelloWorld.main(`*args de linha de comando*`)`. Esta notação `Classe.método()` é usada para se referir a *funções públicas* (métodos declarados como `public` e `static`).

Vamos mostrar mais dois exemplos que você pode compilar e rodar. É recomendável que você os digite e compile para ir se familiarizando com o seu ambiente de desenvolvimento.

HelloWorld Gráfico

Esta outra aplicação faz o mesmo que a anterior mas imprime as palavras "Saudações Javanesas!" em uma janela aberta na tela. É uma aplicação escrita para usar a interface gráfica (GUI). Tudo está dentro da declaração de classe `HelloWorld2`, exceto uma declaração "`import`" que importa a biblioteca onde estão as classes básicas do sistema de janelas (desta vez ela é necessária). Apenas as classes do pacote `java.awt` estão em negrito. Observe que o programa consiste de dois arquivos!

```
// HelloWorld Gráfico – classe que define a interface (não executável)
import java.awt.*;
public class HelloWorld2 extends Frame {
   private Label hello;
```



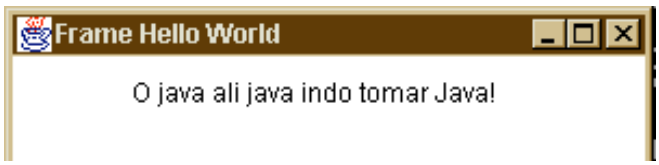
```
   public HelloWorld2(String nome) {      // Isto é um construtor
      super(nome);
      hello = new Label("O java ali java indo tomar Java!");
      FlowLayout flow = new FlowLayout(); // cria política de layout
      this.setLayout(flow);               // aplica política de layout
      this.add(hello);           // add() obedece pol. de layout atual
   }
}
```

```
// classe executável que cria objeto do tipo HelloWorld2

import HelloWorld2;  // importa do CLASSPATH (que inclui diretório atual)

class WinHello {

      public static void main(String[] args) {
            HelloWorld2 javali;
            javali = new HelloWorld2("Frame Hello World");
            javali.setSize(300, 100);
            javali.setVisible(true);
     }
}
```

Detalhes sobre a implementação do programa acima serão vistos no decorrer do curso, nos capítulos sobre applets e sobre programação da interface gráfica. Observe que a primeira classe tem um método chamado `HelloWorld2()` (o mesmo nome que a sua classe). Esse método especial chama-se *construtor* e é utilizado para inicializar objetos quando são criados. E é só isso que o `main` faz aqui: cria um objeto.

No exemplo anterior você podia utilizar qualquer nome para o seu arquivo `*.java`. Neste exemplo, a classe `HelloWorld2` não compila a não ser que o arquivo `.java` se chame `HelloWorld2.java` (lembre-se do `H` e `W` maiúsculos!). Isto é necessário porque a classe foi declarada pública (para que possa ser usada por outras classes fora do diretório atual). O javac exige que só haja uma classe pública por arquivo-fonte. Para garantir isto, exige que o nome do arquivo-fonte seja o mesmo da classe.

Se você rodar o programa (a classe executável `WinHello`) verá saltar na tela uma janela do *Windows* com o texto "O javali já vai indo tomar Java!" dentro dela, como na figura abaixo:



Java chama esse tipo de janela de "`Frame`" e essa aplicação, como qualquer outra aplicação de janelas, é um "`Frame`". Se ela "é", é porque estende `Frame` como uma subclasse (os detalhes sobre orientação objetos serão abordados mais adiante). Isto está representado na declaração da classe `HelloWorld2` através da palavra reservada "`extends`". Este é novamente um exemplo da reutilização. Nós construímos em cima de uma estrutura já existente: o Frame, e com isto fazemos, com poucas linhas, uma aplicação gráfica.

No nosso exemplo um programa executável usou uma classe que nós mesmo criamos (`HelloWorld2`) para construir um *objeto* (`javali`) chamar seus métodos (para torná-la visível e definir seu tamanho). Os métodos de `HelloWorld2` não foram definidos por nós. Nós apenas temos um método que é o construtor, e este foi chamado usando `new` no `main()` do executável. De onde vêm, então, os métodos `setSize()` e `setVisible()`?

A resposta pode estar na herança. Se podemos usar um método que não está disponível na nossa classe, certamente este método foi herdado de uma classe mais acima na hierarquia! Procure na documentação Java e tente descobrir em que classe os métodos `setSize()`, `setVisible()`, `setLayout()` e `add()` estão definidos. A palavra `this` é um ponteiro para o objeto corrente da própria classe. `this.add()`, portanto, chama um método que faz parte dos métodos de `HelloWorld2`.

HelloWorld Applet

Uma terceira versão do "Hello World" é um programa que roda como uma applet, dentro de uma página Web (todas as classes em negrito):

```
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorld3 extends Applet {

    private String mensagem;

    public void init() {
        setBackground(java.awt.Color.white);
    }

    public void paint(Graphics g) {
        mensagem = getParameter("msg");
        g.setColor(java.awt.Color.red);
        g.drawString(mensagem, 20, 20);
    }
```



```
}
```

Novamente, se você tentar rodar este programa a partir da linha de comando, o interpretador acusará um erro, pois não encontrará o método `main`. Esta versão não contém método `main` porque é um applet e só pode ter a sua execução iniciada por um browser. O applet contém métodos especiais que regulam o seu ciclo de vida, eventos, apresentação, etc. como veremos no capítulo sobre applets mais adiante. Um deles, chamado `paint()`, é invocado automaticamente pelo browser e fornece um contexto gráfico (espaço na tela do browser) onde se pode desenhar linhas, definir cores e fontes, e imprimir texto como no programa acima. O método `init()` é chamado uma vez para inicializar o applet. No nosso programa, nós *sobrepusemos* os dois métodos, isto é, definimos uma nova funcionalidade para eles. A *assinatura* (tipo de retorno, nome do método, número e tipo dos argumentos) dos dois tem que ser igual à assinatura dos métodos originais para realizar a sobreposição. Só assim o browser saberá quem chamar e quais parâmetros passar. Quando o browser chama `paint()`, automaticamente passa como argumento um contexto gráfico para o applet.

Assim como o programa anterior era um "`Frame`", este *é um* "`Applet`". Todo applet é uma subclasse de `java.applet.Applet`. A classe `Applet`, está organizada abaixo de uma biblioteca (pacote) chamada de "`java.applet`". `Frame` e `Label` são classes que pertencem à biblioteca "`java.awt`". Usa-se então "`import`" no início do programa para informar a localização das classes utilizadas.

Como vimos, o programa acima não roda sozinho. Applets não são aplicações independentes; precisam de um outro programa que os inicie. Para utilizar a applet é necessário "chamá-la" através de uma página Web usando um descritor HTML `<applet>` como mostrado a seguir:

```
<html>
  <head>
       <title>O Famigerado Hello World em forma de Applet</title>
  </head>

  <body bgcolor=white>
       <h1 style="font-color: red; font-family: sans-serif">
       O Famigerado Hello World...</h1>
       <p>... em forma de Applet!</p>
       <p align=center><hr>
```



```
        <applet code=HelloWorld3.class height=50 width=400>
          <param name=msg value="Leve-me ao seu mestre, terráqueo!">
        </applet>
      </p>
  </body>
</html>
```

O programa lê parâmetros da página HTML através de um outro descritor HTML `<param>`, ou seja, o texto da applet pode ser mudado pelo autor da página HTML que não precisa entender nada de Java.

Digite o programa da applet e salve-o em um arquivo com o nome `HelloWorld3.java`. (novamente, é importante que o programa-fonte tenha *o mesmo nome* que a classe). Depois carregue o HTML em seu browser (*Netscape*, *Internet Explorer*, *HotJava* ou outro que suporte Java 1.1 em diante) ou isoladamente, usando o *AppletViewer* – programa distribuído com o JDK. Para rodá-lo digite:

`appletviewer` *`nome-da-página`*`.html`

Onde `nome-da-página.html` é o nome que você deu ao arquivo HTML onde está a chamada para a sua applet. A figura ilustra a applet acima rodando dentro de uma página em um browser.

1.3. Análise dos programas

Pudemos perceber que o compilador Java cria um arquivo de classe (`.class`), formado por instruções de *bytecodes* (linguagem de máquina virtual Java), para *cada classe* declarada no programa fonte. O programa fonte também é chamado de *unidade de compilação*. Nos nossos exemplo, apenas um arquivo de classe por unidade de compilação. Se uma única unidade de compilação tiver 10 classes declaradas, o compilador produzirá 10 arquivos *bytecode*. Cada classe irá dar origem a um arquivo de mesmo nome, com a extensão `class`.

Há, porém, uma restrição quanto ao nome que pode ser dado ao programa fonte. A linguagem Java estabelece que só poderá haver *uma* classe pública (declarada com o modificador "`public`") em cada unidade de compilação. Existindo tal classe pública, o nome do arquivo fonte deverá ser o *mesmo* que o nome dessa classe, acrescentado naturalmente da extensão `.java` (veja como exemplo a classe `HelloWorld3` que poderia ser compilada no mesmo arquivo que `WinHello`, que não é pública).

Por exemplo, suponha que uma unidade de compilação possui várias classes declaradas `Cachorro`, `Gato` e `Rato` e uma classe *pública* chamada `Animal`. O arquivo fonte deverá chamar-se necessariamente `Animal.java`. Se for usado outro nome o compilador irá exibir uma mensagem de erro e não realizará a compilação. Usando-se o nome correto, o resultado será o esperado: quatro arquivos de classe: `Animal.class`, `Cachorro.class`, `Gato.class`, e `Rato.class`.

Normalmente, as classes que definem programas são públicas, pois desejamos que elas possam ser executadas fora do seu diretório de origem (o que não seria possível se não fossem públicas). Não usamos `public` nos nossos executáveis mais por questões didáticas (para mostrar que o nome da classe deriva da declaração "class" e não do nome do arquivo).

O Método main() e outros métodos

O método `main()`, que está presente nas classes executáveis, sempre tem a sintaxe a seguir:

```
public static void main (String[] args) { ... }
```

Modificadores | Tipo de Retorno | Tipo do Argumento | Argumento | Implementação do Método

Qualquer definição de método Java (veja também os métodos `paint()` e `init()` nos exemplos acima) tem uma estrutura semelhante ao `main()`. Entre as chaves **`{`** e **`}`** está *toda* a *implementação* do método (instruções). É o único lugar[2] do programa onde pode existir um procedimento algorítmico. Após o *nome do método* sempre há um par de *parênteses* que podem ou não conter um ou mais argumentos. O *tipo* de cada argumento sempre é declarado e a variável do argumento pode ser usada apenas localmente. Os argumentos são separados por vírgulas:

```
public int calcula(int a, int b, double limite) {
    int z = a + b;    // z, a e b são variáveis locais
    (...)
    return z;      // z é int!
}
```

O método acima retorna um valor inteiro (tipo `int`). Caso ele não retorne algum valor, o valor deve ser `void`. Isto tem que ser declarado antes do *nome* de cada método. O uso de modificadores é opcional e pode afetar a qualidade ou acessibilidade de um método.

No caso do `main`, os modificadores (neste caso, apenas, obrigatórios) são `public` e `static`. Como `main` não retorna valor, ele é declarado `void`. Ele também pode receber como argumento um *vetor de objetos* do tipo `String`, que chamamos, neste exemplo, de `args`.

Os colchetes `[` e `]` indicam que `args` é um *vetor*. `String` é uma classe que pertence a biblioteca fundamental da linguagem Java e caracteriza uma cadeia de caracteres. Toda classe define um *tipo de dados*. A variável `args`, então, pode conter um vetor de cadeias de caracteres, que podem ser palavras, por exemplo. No caso específico do `main`, `args` é utilizado pelo interpretador para armazenar as palavras digitadas após o nome do programa na linha de comando.

Variáveis e campos de dados

Dentro do método `calcula()` que aparece como exemplo acima, a variável `z` é declarada como sendo do tipo int. Variáveis declaradas dentro de métodos são locais àqueles métodos e só existem dentro do bloco formado pelas chaves.

[2] Há algumas exceções mas não convém mencioná-las no momento.

Modificadores Tipo da Variável valor de inicialização (definição)

Em geral, não são precedidas de modificadores. O mesmo ocorre com as variáveis declaradas como argumentos do método.

```
private static int valor = 15;
```

Variáveis também podem ser declaradas e inicializadas fora dos métodos. Neste caso elas passam a ser chamadas de campos de dados pois definem propriedades do objeto ou classe à qual pertencem. Quase sempre têm modificadores. Métodos (ou construtores) ou variáveis que são declarados com o modificador private só podem ser usados dentro da mesma classe. Se forem declarados com modificadores public, pode ser usados de uma outra classe.
Uma *declaração* de variável tem, portanto, a seguinte sintaxe:

```
modificadores tipo nome_da_variável;
```

A *definição* de uma variável ocorre através de uma expressão de atribuição:

```
variável = expressão;
```

que consiste em se atribuir o resultado da expressão que está do lado direito ao nome de variável que está do lado esquerdo. A expressão do lado direito pode ser uma expressão aritmética, booleana ou um método que retorne um tipo de dados compatível com o tipo declarado para a variável.

Toda instrução simples, seja declaração, atribuição ou invocação de método deve, necessariamente, terminar em ponto-e-vírgula. Expressões mais complexas que consistem de várias instruções simples, como as expressões de controle de fluxo, devem ser organizadas dentro de blocos, delimitados por chaves ({ }). A tabela abaixo lista estes e outros separadores usados em Java:

SEPARADOR	DESCRIÇÃO
(...)	separa blocos de argumentos em métodos
[...]	define vetores (arrays)
{ ... }	separa blocos de código
,	separa argumentos ou variáveis em uma declaração
;	separa linhas de código (marca o final)

As variáveis que são declaradas no corpo da classe não são as únicas usadas pelo programa, mas são as únicas que podem ter acesso externo e serem manipuladas pelos métodos. São chamadas de *variáveis de instância* (pertencem aos

objetos criados com aquela classe), se não tiverem modificador `static`; ou *variáveis de classe*, se tiverem modificador `static` (este modificador será explicado mais adiante).

As variáveis que não forem declaradas `static` pertencem a objetos criados por essa classe e não podem ser usadas dentro de métodos também declarados como `static` sem que isto ocorra através de um objeto.

```
public class Teste {
  public int x;          // pertence aos objetos criados com Teste
  public static int y;   // pertence à classe Teste
  public metodo() {
    int z = y + 1; // OK pois y é static!
    int k = x + 1;  // ERRO! variável não está disponível na classe!
    Teste t = new Teste();  // cria objeto desta classe
    int k = t.x + 1;        // usa x através de t.
}
```

Impressão na saída padrão

No nosso primeiro programa aparece o método:

```
System.out.println("cadeia de caracteres");
```

Em Java, pontos "`.`" são usados para separar classes e objetos de seus campos de dados e métodos. Vimos que `System` (`java.lang.System`) é uma classe da biblioteca fundamental da linguagem Java. `System.out` é uma variável pública de `System` que é um *objeto* que representa a saída padrão do seu computador. Objetos podem ter métodos. É o tipo do objeto (ou a classe) que define esses métodos. O método `println()`, portanto, é método definido na classe que criou out. Qual é esta classe? Procure na documentação! (seção a seguir).

Comentários

Há três formas de se representar comentários em Java:

`/* Texto */`

Comentários estilo C. Todo o texto após o "`/*`"e antes do "`*/`" é ignorado pelo compilador, inclusive novas-linhas. Não podem conter outros blocos de comentário do mesmo tipo, já que o comentário termina assim que encontra o primeiro "`*/`".

Exemplos de uso:


```
for (x /*x é um índice */ = 0; x < 5; x++);
```

Correto. O compilador irá ver: `for (x = 0; x < 5; x++);`

```
s = texto + /* plural */ "s";
```

Correto. Para o compilador: `s = texto + "s";`

```
/* s = texto + /* plural */ "s"; */
```

Errado. O compilador vai ler: `"s"; */`

`// Texto`

Comentários estilo C++. Todo o texto que segue o "`//`" é ignorado até o final da linha.

Exemplo:

```
// s = texto + /* plural */ "s";
```

Correto. O compilador ignorará toda a linha.

`/** Texto */`

Comentários especiais para geração de documentação. São idênticos aos comentários estilo C. O compilador ignora o bloco da mesma forma, mas um programa especial que faz geração automática de documentação, o *Javadoc* (parte do JDK) é capaz de extrair seu texto, comandos de escape especiais (iniciados com "`@`") e adicioná-los a um documento HTML. Além dos comentários, o documento HTML gerado contém outras informações extraídas da classe, como superclasses, sintaxe dos construtores, métodos e variáveis.

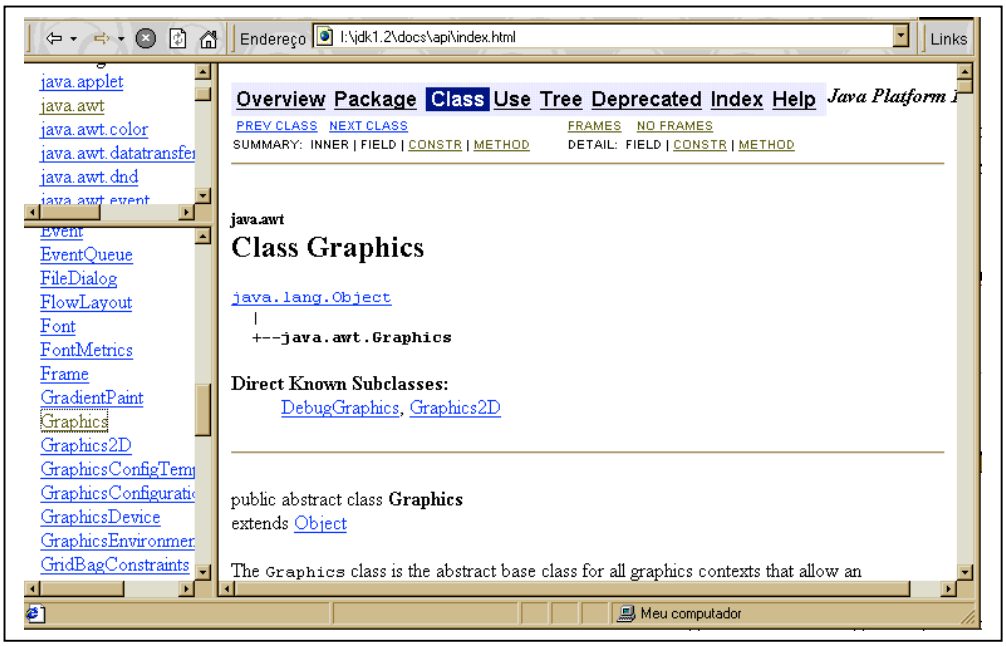
Vamos ver a geração de documentação na prática. Rode o `javadoc` no arquivo `HelloWorld3.java`. Ele vai gerar um arquivo `HelloWorld3.html`, entre outros. Abra esse arquivo no seu browser e veja seu conteúdo.

```
C:\> javadoc HelloWorld3.java
Generating packages.html
generating documentation for class HelloWorld3
Generating index
Sorting 5 items . . . done
Generating tree
```

A documentação gerada tem o mesmo estilo que a documentação da API Java. Se você acrescentar comentários de documentação antes dos métodos, eles aparecerão no HTML como descrições do seu código. Vamos dar uma olhada na documentação em hipertexto. É essencial que você saiba usá-la.

1.4. Como consultar a documentação

A documentação da linguagem Java é distribuída em um pacote à parte do JDK. Consiste de dezenas de milhares de arquivos interligados por hipertexto. Toda a documentação foi gerada com o Javadoc sobre os códigos fonte das classes da API Java. Para abrir a documentação, abra o arquivo `index.html` do subdiretório `/jdk1.2/docs/`. A partir dessa página, que possui vários tutoriais e outras explicações sobre os recursos de Java, procure o link *API & Language*, depois *Java Platform 1.2 Specification*. Você deve encontrar uma página como a seguinte:



A janela se divide em três partes. A do canto superior esquerdo permite que você selecione um pacote da API Java (`java.awt`, por exemplo). Na janela logo abaixo, você pode selecionar uma classe desse pacote (`Graphics`, por exemplo). Quando você fizer isto, na janela principal aparecerá uma página descrevendo todos os métodos, variáveis e construtores da classe selecionada, mostrando uma descrição e a hierarquia de classes em detalhes.

Experimente e explore a documentação. Aprenda a utilizá-la sempre e principalmente enquanto estiver aprendendo Java.

Exercícios

1. Pratique mais com os programas apresentados neste capítulo. Modifique linhas do código e veja os erros que você obtém. É importante se familiarizar com o ambiente de desenvolvimento antes de continuar.
2. Localize o método `drawString()` da classe `java.awt.Graphics` (veja terceiro exemplo) na documentação Java. Veja quais outros métodos a classe Graphics oferece. Tente usar outros como `fillRect()` e `drawOval()`. Veja como funciona o método `setColor()` e tente mudar a cor dos objetos desenhados.
3. Todas as classes que compõem a biblioteca fundamental da linguagem Java são escritas em Java. Se você instalou o JDK, pode encontrar as listagens do código-fonte de todas as classes no seu sub-diretório `src` (por exemplo `c:\java\src` ou `/java/src`). Veja, por exemplo, as listagens de `java/awt/Frame.java`, `java/awt/Label.java` e `java/applet/Applet.java`.
4. Tente entender como funciona o comando `System.out.println()`. Use a documentação e procure na classe `System` pela variável pública `out`. O que ela significa? Ela é uma referência para um objeto de um determinado tipo (ou classe). Que tipo? Qual a classe que foi usada para formar o objeto `out`? Onde está o método `println()`?
5. Descubra como usar o objeto `in` para ler texto da entrada padrão. Escreva um programa que leia uma linha da entrada padrão e escreva a mesma linha na saída.

Recapitulação

Verifique se você aprendeu os principais tópicos deste capítulo:

- Descreva as principais características da linguagem Java.
- Descreva o funcionamento da máquina virtual Java e da coleta de lixo.
- Defina pacote, classe e método.
- Escreva, compile e execute uma aplicação.
- Escreva, compile e execute um applet.
- Localize o método de uma determinada classe (por exemplo, `add()` de `Frame`) na documentação da linguagem.

JAVA 2

Sintaxe e Estrutura

ESTE MÓDULO APRESENTA A SINTAXE E ESTRUTURA DA LINGUAGEM JAVA, os operadores, expressões de controle de fluxo e a criação, declaração e utilização de vetores em Java. Destaca também as diferenças entre Java e outras linguagens.

Tópicos abordados neste módulo

- Estrutura de um programa em Java
- Sintaxe: comentários, instruções, identificadores, literais, tipos, palavras-chave.
- Convenções de programação
- Estruturas de dados
- Criação e declaração de tipos novos
- Operadores
- Expressões de controle de fluxo
- Vetores

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- Identificar as palavras-chave da linguagem Java
- Listar os oito tipos primitivos
- Saber criar um novo tipo de dados com uma classe
- Entender o que é uma classe, um método, uma variável membro e uma referência.
- Construir um novo objeto com new
- Distinguir variáveis de instância de variáveis locais
- Identificar e saber usar os operadores de Java
- Saber realizar conversões de tipos
- Saber usar as expressões de controle de fluxo
- Saber inicializar, criar, usar e copiar vetores.

1.1. Estrutura e sintaxe

As estruturas básicas de uma unidade de compilação Java: as declarações de classe (`class`), de pacote (`package`) e importações (`import`) devem aparecer no código respeitando uma determinada ordem. Se houver uma declaração `package`, do tipo:

```
package acme.java.awt;
```

ela deve ser a primeira estrutura do arquivo. Apenas comentários podem precedê-la. Essa declaração afirma que a classe (ou as classes) do arquivo de código fonte onde está contida pertencem ao referido pacote. Se um pacote não

for definido, a classe pertencerá a um pacote global que restringe-se ao diretório atual onde a classe irá executar (ela só será capaz de ser usada por outras classes que estão no mesmo diretório e que também não definam pacote).

Se houver declarações `import`, elas devem seguir a declaração `package`. A instrução `import` informa a localização de classes e não de pacotes. Por exemplo, as instruções

```
import java.awt.*;
import java.awt.event.*;
```

tornam acessíveis ao código todas as classes dos pacotes `java.awt` e `java.awt.event`. Importar todas as classes de `java.awt` não importa também o subpacote, por isto, é preciso importar suas classes explicitamente.

Após o `package` (opcional) e os `imports` (também opcionais), podem vir uma ou mais declarações e implementações de classe. A declaração deve começar com `class` ou `public class`. Se começar com `public class`, o nome do arquivo fonte (`.java`) deve ter o mesmo nome que a classe e não poderá existir outra classe no mesmo arquivo declarada com o modificador `public` (apenas classes iniciando com `class` e não `public class`). Existe em java um tipo especial de classe chamada de *interface*. A declaração poderá então iniciar com `interface` ou `public interface`, com as mesmas regras quando ao nome do arquivo fonte.

Além de `package`, `import` e `class` (ou `interface`) não poderá haver mais nada no primeiro nível do arquivo fonte Java a não ser comentários. Métodos, construtores, variáveis e constantes só podem aparecer dentro das classes e quaisquer procedimentos algorítmicos só podem estar presentes dentro de métodos, construtores e blocos `static` (um bloco especial para inicialização de valores estáticos).

As seções a seguir detalharão aspectos da sintaxe da linguagem Java e construções que poderão ser usadas nas estruturas mencionadas.

Comentários

Os comentários em Java já foram apresentados no capítulo anterior. Podem ser de dois tipos:

comentários de uma linha:

```
// isto é um comentário
```


comentários de bloco:

```
/*
 * Isto é um comentário
*/
```

Os comentários de bloco podem ainda ser usados para gerar documentação. Para isto, é necessário que sejam usados antes de classes, métodos, construtores ou variáveis e que iniciem o bloco com dois asteriscos:

```
/**
 * Este é um <i>comentário</i> de documentação.
*/
```

Comentários desse tipo aceitam vários comandos geradores de estruturas HTML e também HTML (como ilustrado acima) dentro do mesmo. Os comandos, o HTML e o texto dos comentários são usados para gerar documentação em hipertexto semelhante àquela disponível para a API (distribuída pela *Sun*). Para isto, é preciso usar a ferramenta `javadoc`.

Blocos e declarações

Uma declaração em Java consiste de uma única linha de código que termina em um ponto-e-vírgula. A declaração (a linha de código) pode ocupar múltiplas linhas do arquivo pois os espaços extras, tabulações e quebras de linha são ignorados. Tanto faz escrever:

```
System.out.println("Resultado: " + (3 * 14 / 19));
```

como escrever:

```
System.out.println
(            "Resultado: " +
        (3 *
                              14 /
               19));
```

O espaço em branco, porém, deve ser usado para tornar o código mais legível e não o contrário. Pode ser usado para endentar os blocos, ou, instruções compostas. Blocos são trechos de código entre chaves `{` e `}`. Variáveis declaradas pela primeira vez em um bloco são locais àquele bloco. Blocos são usados após declarações de classes, métodos, construtores, estruturas de controle de fluxo, etc.:


```
public class UmaClasse {
    public void umMetodo() {
       if(true) {
             System.out.println("É true!");
       }
    }
}
```

Use o espaço em branco para endentar as instruções dentro de blocos, deixar espaços verticais entre métodos e trechos longos de código, deixar espaços verticais em expressões aritméticas, com o objetivo de deixar o código mais legível.

Identificadores

Identificadores em Java são os nomes que usamos para identificar variáveis, classes, e métodos. Não fazem parte da linguagem e são criados arbitrariamente pelo programador.

Pode-se usar qualquer letra ou dígito do alfabeto Unicode, ou os símbolos "$" e "_" (sublinhado) em um identificador. Dígitos podem ser usados como parte de identificadores desde que não sejam o primeiro caractere.

Identificadores em Java podem ter qualquer tamanho. Não há limitação em relação ao número de caracteres que podem ter, porém deve-se evitar nomes muito grandes para identificar classes, uma vez que também são usados como nomes de arquivo. Palavras reservadas (veja adiante) não podem ser usadas como identificadores.

Como você já observou, Java distingue letras maiúsculas de minúsculas. Isto também vale para palavras usadas como identificadores. Valor é diferente de valor que é diferente de VALOR.

Os seguintes identificadores são legais em Java:

```
Cão    variável    CLASS       R$13  índice      αντηροπος
```

Mas não são recomendadas pois podem confundir. Já pensou se você usa a palavra `índice` como variável e depois, por engano, usa `indice`? É um erro difícil de achar. `CLASS` não é palavra reservada, mas `class` é. Evite confusões! Usar **α**ντηροπος pode ser interessante se você fala grego mas imagine se você usa tais caracteres em um nome de classe executável. Como você vai conseguir rodar o programa se não tiver as fontes correspondentes no seu sistema?

Esses outros identificadores são ilegais:

```
ping-pong    Johnson&Johnson        R$13.00    2aParte
```

Palavras-chave

As palavras seguintes, junto com os valores `true` e `false`, que são literais booleanas, são reservadas em Java. Não podem ser usadas para identificar classes, variáveis, pacotes, métodos ou serem usadas como nome de qualquer outro identificador.

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

Apesar de reservadas, as palavras `const`, e `goto`, e não são usadas atualmente (Java 2).

Os nomes de classes fundamentais da linguagem não são consideradas palavras reservadas, mas você deve evitar usá-las. Se você cria uma variável chamada `String` você terá erros de tipo incorreto. Se criar uma classe chamada `String` terá erros de ambigüidade a não ser que o seu programa crie um pacote específico para essa sua classe. Se puder evitar, porém, evite usar nomes de classes dos pacotes Java.

Literais

Literais são valores. Podem ser valores numéricos, booleanos, caracteres individuais ou cadeias de caracteres. Literais podem ser usados diretamente em expressões, passados como argumentos em métodos ou atribuídos a uma variável. Exemplos de literais são:

- `12.4` (12,4 ponto flutuante decimal)
- `0377` (377 inteiro octal)
- `0xff991a` (FF991A inteiro hexadecimal)
- `true` (literal booleana)

- `'a'` (caractere)
- `"barata"` (cadeia de caracteres)

Numéricos

Os literais numéricos são representados em formato decimal, por *default*. Se um literal numérico for precedido por um zero, será considerado um número octal, e se for precedido por `0x`, será interpretado como um número hexadecimal. Veja alguns exemplos de literais numéricos:

```
12.4  0xab779c .27777e-23      0137  20  Double.NaN
```

Literais numéricos variam em faixa e representação dependendo do *tipo* de dados das variáveis usadas para armazená-los. Veja na seção seguinte as faixas de valores suportados para cada tipo.

Booleanos

Os literais booleanos são apenas dois: `true` e `false`. Representam uma condição verdadeira ou falsa.

Caracteres e strings

Os literais de caracteres podem ser representados através de um caractere isolado entre aspas simples (por exemplo: `'a'`, `'Z'`) ou usando uma seqüência de escape especial (veja adiante), também entre aspas simples. Veja alguns exemplos:

```
'H'    '\n'   '\u0044'  '\u3F07'  '1'  '\\'
```

Apesar do tipo `String` não ser um tipo primitivo em Java (`String` é uma classe que representa objetos do tipo "cadeia de caracteres"), Java define literais do tipo `String` formados por conjuntos de caracteres entre aspas duplas. Alguns exemplos:

```
"anta"  "vampiros são morcegos"     "" (vazia)
"\u3F07\u3EFA \u3F1C"               " " (espaço)
"Uma linha\nDuas Linhas\tTabulação"
```

Qualquer caractere em Java pode ser representado usando o padrão Unicode, de 16 bits, que permite representar 65536 caracteres diferentes. Além disso, as seqüências de escape listadas na tabela a seguir podem ser usadas para representar certos valores especiais que podem aparecer em uma literal do tipo `char` ou `String`.

SEQÜÊNCIA	VALOR DO CARACTERE
\b	Retrocesso (backspace)
\t	Tabulação
\n	Nova Linha (new line)
\f	Alimentação de Formulário (form feed)
\r	Retorno de Carro (carriage return)
\"	Aspas
\'	Aspa
\\	Contra Barra
nnn	O caractere correspondente ao valor octal *nnn*, onde *nnn* é um valor entre 000 e 0377.
\u*nnnn*	O caractere Unicode *nnnn*, onde *nnnn* é de um a quatro dígitos hexadecimais. Seqüências Unicode são processadas antes das demais seqüências.

Os caracteres de escape Unicode são formas de representar caracteres que não podem ser exibidos em sistemas que não suportam Unicode ou que não possuem as fontes corretas para exibir os caracteres. Por exemplo, em um terminal Unicode, você pode ver na tela os ideogramas Higarana e Katakana (japonês) e vários outros que correspondem à faixa `\u3040` a `\u9FFF` do código Unicode. Em um sistema ASCII, Java representa esse caractere usando o escape Unicode \u*nnnn*.

Você pode usar identificadores com acentos e cedilha para dar nomes a qualquer variável, método ou classe, pois Java os converte para Unicode antes de compilar (porém evite fazer isso no caso dos nomes das classes, já que os nomes de arquivo gerados são dependentes do sistema operacional).

Para representar aspas e contra-barras dentro de um literal String ou de caractere, é necessário usar um escape também, precedendo o caractere por uma outra contra-barra, por exemplo, para imprimir a linha:

```
"Doom II", C:\games\doom
```

é preciso usar:

```
System.out.println("\"Doom II\", C:\\games\\doom");
```


Tipos primitivos

Java acrescenta `byte` e `boolean` ao conjunto de tipos de dados primitivos das linguagens C e C++. A principal diferença em relação a C ou C++ é que os *tamanhos* em bytes dos tipos em Java são definidos, e não dependem da plataforma, o que não acontece naquelas duas outras linguagens. Além disso, em C, quando uma variável é usada sem ser inicializada, geralmente contém lixo. Java define valores *default* para todos os tipos, que são usados caso uma variável não tenha sido inicializada na criação do objeto a qual pertence.

Variável alguma em Java tem permissão para ter um valor não definido. Variáveis locais têm que ser inicializadas antes de serem usadas ou um erro de compilação irá ocorrer. A primeira inicialização de uma variável local também não pode ser feita dentro de um bloco `if` a não ser que ela própria tenha sido declarada dentro desse bloco (é local ao `if`). Se isto ocorrer, o compilador acusará o erro mais adiante quando a variável for usada e dirá que "a variável pode não ter sido inicializada). Variáveis definidas dentro do bloco de uma declaração de classe (membros – de instância e de classe) são inicializadas com valores *default*.

A tabela abaixo relaciona os tipos primitivos em Java, seus tamanhos, faixa de valores dos seus literais e valores *default*.

TIPO DE DADOS	TAMANHO	VALORES	DEFAULT
`boolean`	8 bits	`true` ou `false`	`false`
`byte`	8 bits	-128 a 127	0
`short`	16 bits	-32768 a 32767	0
`char`	16 bits	\u0000 a \uffff	\u0000
`int`	32 bits	-2147483648 a 2147483648	0
`long`	64 bits	-9223372036854775808 a 9223372036854775807	0L
`float`	32 bits	1,40239846e-45 a 3,40282347e38	0.0f
`double`	64 bits	4,94065645841264544e-324 a 1.79769313486231570e308	0.0d

Qualquer identificador válido em Java pode seru usado para dar nome a uma variável. Todas as variáveis em Java devem ser *declaradas* antes de serem usadas. Se uma variável for declarada como sendo de um determinado tipo receber um valor de outro tipo, ela poderá ter que ser convertida explicitamente

através de uma conversão forçada (*cast*), também chamada de coerção. Variáveis de tipos menores podem, em geral, terem seus valores atribuídos a tipos maiores sem coerção. O contrário exige que o tipo (resultante) seja informado entre parênteses. Se o valor armazenado for maior que o que o tipo final pode comportar, haverá perda de informação.

```
float f;
int i;

f = 3.14;
i = (int)f; // i recebe f, que é convertido para int
            // (números depois da vírgula são truncados)
```

A conversão ou coerção só é possível entre tipos numéricos e caracteres. Booleanos são inconversíveis. Uma discussão maior sobre conversão e coerção será apresentada mais adiante.

Tipos boolean

Valores booleanos não são inteiros e nem podem ser tratados como tal. Também não é possível converter valores booleanos em outros tipos através de atribuição. Para traduzir valores booleanos em inteiros, pode-se usar um bloco `if` ou o operador ternário:

```
boolean b; int i;
b = (i != 0); // converte 0 para false e !0 para true
i = b ? 1 : 0; // converte true para 1 e false para 0
```

Tipos char

Tipos `char` contém um caractere Unicode de dois bytes. O primeiro `byte` corresponde ao ASCII ou ISO-Latin1.

A conversão de `char` em `int` ou `long` obtém o valor Unicode do caractere. Os tipos `char` não tem sinal. Se um `char` (16 bit) for convertido para um tipo `byte` (8 bit) ou `short` (16 bit), pode resultar em um valor negativo.

Um `char` sempre pode ser manipulado como um `int`. Se for necessário representar o caractere, porém, é preciso convertê-lo (através de um cast) para `char`:

```
int letra = 'A';
System.out.println("O código Unicode da letra "
                   + ((char)letra) + " é " + letra);
```


Tipos inteiros

Todos os tipos inteiros tem sinal (não existe a palavra-chave `unsigned` como em C). Inteiros podem ser do tipo `byte` (8 bit), `short` (16 bit), `int` (32 bit) ou `long` (64 bit). A coerção de tipos de tamanhos maiores em tipos menores pode resultar em perda de dados caso os valores a serem convertidos sejam maiores que os valores máximos armazenados por cada tipo.

A representação *default* para inteiros é `int`. Se uma atribuição para uma variável do tipo `long` recebe um `int`, este é automaticamente convertido (e ampliado) para long. Se o literal for um número maior que o valor máximo de um `int`, porém, o número será `long` e provocará erro ao ser atribuído a uma variável do tipo `int` a não ser que seja através de uma coerção (que provocará perda de informação). Pode-se distinguir literais do tipo `long` das constantes `int` utilizando o caractere "L" como sufixo (ex: `127L`). Evite, porém, usar o "`L`" minúsculo (é permitido, mas como ele se parece muito com o algarismo "`1`", poderá ser fonte de confusão).

A divisão de um número inteiro por zero sempre causa uma exceção (erro em tempo de execução) do tipo `ArithmeticException`.

Tipos de ponto-flutuante

Qualquer valor numérico contendo um ponto decimal ou um expoente (caractere "e" seguido de um número) é um literal de ponto flutuante. São sempre do tipo `double` (64 bit – dupla precisão) por *default*. Tipos `float` (32 bits – precisão simples), quando inicializados, devem conter o sufixo "F" pois sem esse sufixo, o literal é considerado double. O sufixo "D" pode, opcionalmente, ser usado para identificar valores de dupla-precisão:

```
float f = 300.0;          // Erro: 300.0 é double e não cabe em float
float f = 300.0f;         // OK pois 300.0f é float
float f = (float) 300.0; // OK. double convertido em float
float f = 300;            // OK. int converte em float sem cast
```

Nenhuma operação de ponto flutuante provoca exceções. Divisões por zero provocam números infinitos ou indeterminações identificadas pelas constantes `Float.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY` e `Float.NaN`, que representam respectivamente, infinito positivo, infinito negativo e "não é número".

Convenções

Java possui várias convenções de codificação que visam facilitar a programação e principalmente a análise do código. Essas convenções são adotadas em toda a documentação. O seu uso é opcional mas recomendado.

Nomes de classes e interfaces

Os nomes de classes são substantivos (são coisas) com caixa mista (cada palavra começa com maiúsculas). Por exemplo:

```
class LivroDeReceitas
class Circulo
```

Nomes de métodos

Os nomes de métodos devem ser verbos (representam ação) com caixa mista (como as classes), mas, sempre com a primeira letra em caixa baixa:

```
public void imprimirRelatorio()
```

Se você encontrar na API Java métodos com a primeira letra maiúscula observe bem e concluirá que é um construtor. Construtores têm sempre o mesmo nome que a classe que o contém (portanto, primeira letra maiúscula) e não possuem tipo de retorno (`void`, `int`, `String`).

Nomes de variáveis

Assim como os métodos, variáveis devem aparecer em caixa mista com a primeira letra minúscula.

```
int moscaVerde;
```

Constantes

Se você encontrar variáveis com todas as letras maiúsculas na API Java, são constantes escalares. As constantes têm todas as letras maiúsculas e se forem formadas por múltiplas palavras, essas palavras devem ser separadas por sublinhado:

```
public final int UMA_CONSTANTE = 15;
```

Constantes não-escalares (objetos) têm caixa mista como as variáveis comuns (pois as suas referências não são constantes). Por exemplo: `red` e `blue` são constantes da classe `Color`.

Nomes de pacotes

Pacotes são representados com todas as letras minúsculas (mesmo que contenham mais de uma palavra)

1.2. Classes e objetos

Estruturas

Suponha que, em determinado programa em Java, você precise especificar um círculo. O círculo pode ser totalmente especificado com um raio e as coordenadas do seu centro. Usando os tipos primitivos de Java, podemos especificar um círculo, dentro de um método, da forma:

```
double x0;
double y0;
double raio;
```

E então, para usar este círculo, bastaria definir cada uma das variáveis:

```
x0 = 1.2;
y0 = 0.8;
raio = 1.5;
```

Assim, poderemos desenhá-lo em uma determinada posição da tela, usando a equação $(x - x_0)^2 + (y - y_0)^2 = raio^2$ e variando x e y.

Suponha agora que o nosso programa exige que lidemos com vários círculos ao mesmo tempo. Teremos que ter três variáveis diferentes para cada círculo e repetir o mesmo processo várias vezes. Só para representar desenhar 100 círculos, teríamos que fazer:

```
double xCirc1, yCirc1, raioCirc1;
double xCirc2, yCirc2, raioCirc2;
(...)
double xCirc100, yCirc100, raioCirc100;
```

e teríamos que especificar cada tipo um por um. Podemos, é claro, usar vetores. A desvantagem é que o vetor provavelmente seria usado para agrupar partes de cada círculo como, por exemplo, todas as coordenadas x, todos os raios, etc. Não poderíamos errar na ordem de um vetor que tinha que sempre estar alinhado com o outro. E se o objeto fosse mais complicado (tivesse mais que três propriedades e de tipos diferentes)? E se estes círculos fossem também

propriedades de um objeto mais complexo? A solução, talvez, seja criar um novo tipo de dados!

Tipos de dados abstratos

Várias linguagens, mesmo as que não são orientadas a objetos, permitem que o programador crie *tipos de dados abstratos*, ou *estruturas*. Em Java, podemos fazer isto com uma *classe*, que é muito mais que uma mera estrutura, como veremos no capítulo seguinte. A estrutura define somente um estado, uma coleção de propriedades. Não define *como* os seus dados podem ser transformados (métodos). Uma classe Java pode conter não só os tipos básicos que a definem, mas também os métodos que caracterizam o comportamento dos objetos criados a partir dela.

Criação de um novo tipo

Para produzir um novo tipo de dados *Círculo* em Java, com as propriedades x_0, y_0 e *raio*, poderíamos definir:

```
public class Circulo extends Object {
       public double    x0,
                        y0,
                        raio;
}
```

Isto é lido da seguinte forma: "Um Círculo *é um* objeto que *tem uma* coordenada x, uma coordenada y e um raio". Lembre-se que a parte `extends Object` é opcional e geralmente é omitida por ser implícita na declaração de classes Java.

Agora, podemos tratar a classe recém-criada como um novo *tipo de dados* e, dentro de um método ou construtor de outra classe (possivelmente uma classe executável) declarar variáveis como *referências* para estes círculos:

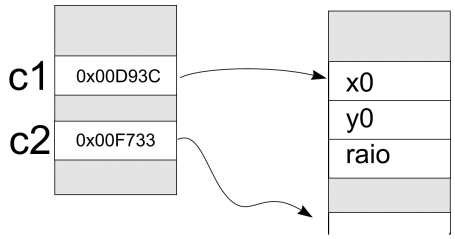
```
Circulo c1, c2, c3;
```

Ao fazer isto, não criamos novos objetos, apenas alocamos memória para as suas *referências* ou *ponteiros*. Estas referências contém o endereço de memória dos objetos. O endereço é definido em tempo de execução e, portanto, não se pode realizar aritmética com ele como em C ou C++.

Criação de um objeto

Para efetivamente *criar* um objeto, é necessário alocar memória para ele usando o operador "new", da forma:

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



Observe que o gerenciamento de memória para os *tipos de objetos* ocorre de forma diferente dos *tipos primitivos*. Na declaração de um `int`, a memória para o seu armazenamento é imediatamente alocada. O sistema sabe que um int ocupa 32 bits. No caso de objetos, a alocação de memória só é possível através do operador `new` que chama um procedimento (construtor) que irá determinar quando espaço será necessário para o objeto e alocá-lo.

Depois de criado o objeto, Os valores iniciais de seus campos de dados são inicializados a zero (ou valores equivalentes, de acordo com o tipo). Somente depois de criado o objeto, podem os campos de cada circulo ser acessados usando o operador "." (ponto):

```
c1.x0 = 1.2;
c1.y0 = 0.8;
double x = c1.x0 + 2;
```

Qualquer classe, portanto, representa um *tipo* de dados e qualquer variável de classe, de instância ou local pode ser declarada como sendo do "tipo" de uma determinada classe da mesma maneira que se declara tipos primitivos. Por exemplo:

```
long preço;  // preço é variável do tipo primitivo long
String nome; // nome é variável do tipo objeto String
Círculo c;   // c é variável do tipo objeto Círculo
```

Referências

Nenhuma declaração *cria* objetos mas apenas *declara* uma variável como tendo o tipo definido em sua classe (`String` ou `Círculo` no exemplo acima), preparando a variáveis para que seja uma *referência* a um objeto desse tipo.

A declaração

```
long preço;
```


estabelece que a variável `preço` está preparada para receber um valor numérico inteiro primitivo do tipo `long` (aloca 64 bits de memória para acomodar o `long`). Após a declaração, é comum ter uma atribuição, *definindo* o valor da variável, por exemplo:

```
preço = 35700;
```

É simples. A criação de um objeto é mais complexa. O sistema não sabe quanto espaço será necessário para acomodar um `Círculo` e então só poderá alocar o espaço necessário para acomodar a referência (32 ou 64 bits).
No nosso exemplo, a alocação da memória para o objeto `Circulo` vai exigir pelo menos 3 vezes 64 = 192 bits, referentes aos campos x, y e raio, sem contar o espaço necessário para os métodos herdados. Também é preciso chamar um construtor que irá dizer como se constrói o objeto. A variável declarada recebe, por atribuição, uma referência (ponteiro) para este objeto:

```
c = new Circulo();
```

`Circulo()` é o construtor e `new`, o operador de alocação de memória usado para criar novos objetos. Depois do `new`, a memória foi alocada, e c é agora uma referência para a posição de memória onde está o objeto que já pode ser usado. A tentativa de usar um objeto antes de sua criação (com `new`) provoca uma exceção em tempo de execução (`NullPointerException`).

Membros de um objeto

As váriáveis e métodos definidos no interior de uma classe são considerados membros de um objeto desde que não possuam o modificador static na sua declaração. Por membros de um objeto queremos dizer que variáveis e métodos fazem parte do objeto criado. Uma classe (tipo) pode ser usada para declarar várias variáveis. Cada uma, porém, ao ser inicializada com um construtor, aponta para um objeto diferente, que possui as variáveis e métodos definidos na classe. Os métodos e variáveis só podem ser usados pelos objetos e não pelas classes. As classes são meras plantas ou moldes usados para moldar ou construir objetos reais que têm propriedades e que podem realizar ações.

Por exemplo, a classe:

```
class UmObjeto {
   private int ox;
   private static int cx;
```



```
    public static void clmet() { ... }
    public void obmet() { ... }
}
```

tem duas variáveis e dois métodos. Os objetos criados com essa classe terão apenas uma variável e um método, que são seus membros. Os outros dois são membros da classe, pois foram declarados com `static`. Só há uma classe, portanto, só há uma cópia da variável `cx`. A variável `ox` e o método `obmet()` só podem ser usados através de objetos. Já o método `clmet()` e a variável cx podem ser usadas de qualquer lugar dentro da classe. Um método da própria classe `UmObjeto` pode criar instâncias (objetos) a partir de si próprio:

```
public static void clmet() {
   UmObjeto obj = new UmObjeto();
   obj.ox = 13;
   obj.obmet();
   cx = 15;
}
```

Um método estático não pode chamar, porém, variáveis de instância pertencentes ao objeto a não ser que crie o objeto como acima:

```
public static void clmet() {
   ox = 13;  // dá ERRO de compilação
   obmet();  // dá ERRO de compilação
   cx = 15; // OK. É static.
}
```

Portanto, `ox` e `obmet()` são *membros* dos objetos criados através da classe `UmObjeto`.

1.3. Expressões e controle de fluxo

Esta seção apresentará os operadores usados na linguagem Java e as estruturas básicas de dados.

Operadores

Operadores permitem a realização de tarefas como adição, subtração, multiplicação, atribuição, etc. Podem ser divididos em três categorias: operadores booleanos, operadores de atribuição e operadores numéricos. Os operadores de Java são quase os mesmos de C/C++, com algumas ausências.

A tabela abaixo lista os operadores usados na linguagem Java:

OPERADOR	FUNÇÃO	OPERADOR	FUNÇÃO
+	Adição	~	complemento
-	Subtração	<<	deslocamento à esquerda
*	Multiplicação	>>	deslocamento à direita
/	Divisão	>>>	desloc. a direita com zeros
%	Resto	=	atribuição
++	Incremento	+=	atribuição com adição
--	Decremento	-=	atribuição com subtração
>	Maior que	*=	atribuição com multiplicação
>=	Maior ou igual	/=	atribuição com divisão
<	Menor que	%=	atribuição com resto
<=	Menor ou igual	&=	atribuição com AND
==	igual	|=	atribuição com OR
!=	não igual	^=	atribuição com XOR
!	NÃO lógico	<<=	atribuição com desloc. esquerdo
&&	E lógico	>>=	atribuição com desloc. direito
||	OU lógico	>>>=	atrib. C/ desloc. a dir. c/ zeros
&	AND	?:	Operador ternário
^	XOR	(*tipo*)	Conversão de tipos
|	OR	instanceof	Comparação de tipos

São vários tipos diferentes de operadores: atribuição, adição, multiplicação, comparação, deslocamento, etc. A ordem em que uma expressão é resolvida depende da precedência dos seus operadores, que é mostrada na tabela a seguir.

Os valores em posição mais alta na tabela têm maior precedência. Na posição horizontal, a precedência é a mesma. A primeira coluna indica a associatividade. D a E significa Direita para Esquerda.

ASSOC	TIPO DE OPERADOR	OPERADOR
D a E	separadores	`[] . ; , () (`
E a D	operadores unários	`new (cast) expr++ expr-- ++expr --expr +expr -expr ~ !`
E a D	multiplicativo	`* / %`
E a D	aditivo	`+ -`
E a D	deslocamento	`<< >> >>>`
E a D	relacional	`< > >= <= instanceof`
E a D	igualdade	`== !=`
E a D	AND	`&`
E a D	XOR	`^`
E a D	OR	`|`
E a D	E lógico	`&&`
E a D	OU lógico	`||`
D a E	condicional	`?:`
D a E	atribuição	`= += -= *= /= %= >>= <<= >>>= &= ^= !=`

Observe que a comparação é realizada com um sinal de "=" duplo. O uso de um único sinal "=" caracteriza uma atribuição.

Também são bastante usados os operadores unários "++" e "−" para incrementar (somar 1) ou decrementar (subtrair 1), respectivamente, uma variável. Estes operadores podem alterar o valor da variável antes ou depois que ela for usada.

Java sobrecarrega o operador "+" para realizar a concatenação de `Strings`, além da adição de valores numéricos.

As expressões a seguir são válidas em Java:

```
 1 + 2
(1 + 2) * i
(a > 0) && (a < 1)
 a == 2
```

As duas primeiras expressões são do tipo de `i` (se `i` for inteiro, o resultado é inteiro). Algumas expressões, porém, podem resultar em conversões de tipos, como vimos anteriormente. A terceira e quarta resultam em `true` ou `false` (resultado tipo `boolean`).

Conversão e coerção

Qualquer tipo numérico pode ser convertido para outro tipo numérico. Apenas os booleanos não são conversíveis de forma alguma. As conversões implícitas só podem ocorrer quando o tipo que recebe a atribuição tem o mesmo tamanho em bits ou precisão, ou mais que o tipo atribuído. No contrário, o compilador irá reclamar.

Para fazer essas conversões "ilegais", pode-se usar o operador de coerção (cast). Com ele, o risco fica com o programador que diz ao compilador que sabe que pode perder dados. Ele diz isto colocando o tipo a ser convertido entre parênteses antes da expressão.

Há quatro tipos de conversões entre tipos primitivos: por atribuição, por passagem de parâmetro em método, por promoção aritmética e por casting. A conversão por *atribuição* é legal quando o tipo do lado esquerdo é maior em número de bytes ou e precisão que o tipo do lado direito:

```
int i = 10;
double d = 12.3;
d = i; // legal, d armazena 10.0
```



```
i = d; // ilegal!!!
```

A conversão por passagem de parâmetro em um *método* é equivalente. Um método declara receber variáveis de um determinado tipo e recebe outro. Se o tipo que o método for receber for menor que o do método, a conversão também é legal.

```
public void agua(int quantidade) { ...}
(...)
    x.agua(char c = A); // OK
    x.agua(float f = 1.1f) {...} // ilegal
```

A conversão por *promoção aritmética* ocorre quando há uma expressão com vários tipos diferentes. Antes da expressão ser calculada, todos os tipos são promovidos para o tipo maior, ou seja, na expressão abaixo, todos serão promovidos para double:

```
int x = 5;
long ab = 10;
double = x + ab + 3.25D;
```

Finalmente, com coerção o programador tem a disposição um meio de converter qualquer tipo. O programador é que terá que resolver sue o resultado é válido.

```
int i = 10;
double d = 12.3;
d = i; // legal, d armazena 10.0
i = d; // ilegal!!!
i = (int)d; // legal, mas trunca o resultado.
```

A promoção aritmética também ocorre em operações entre bytes e shorts. Com estes valores, eles são sempre promovidos para int antes de operarem.

```
int i = (byte) (b1 + b2); // b1 e b2 são do tipo byte.
```

Controle de fluxo

As expressões de controle de fluxo são basicamente as mesmas do C ou C++. Uma diferença fundamental é que os resultados das expressões usadas como teste devem ser obrigatoriamente booleanas (resultar em `true` ou `false`).

Seleção if ... else

Sintaxe básica:

```
if (expressão) {...}
[else if (expressão) { .. }]
[else {...}]
```

O primeiro `if` é obrigatório. Os outros blocos são opcionais. Se a expressão entre parênteses resultar em `true`, o código entre `{` e `}` será executado, caso contrário o bloco `else` é executado, se existir. *Exemplo:*

```
if (valor == 0) {
   fatorial = 1;
} else {
   fatorial = valor;
   ...
}
```

Seleção while, do...while

Sintaxe básica:

```
while (expressão) {...}    ou
do {...} while (expressão);
```

Se o resultado da expressão for `true`, o bloco de `while` será executado. Na seleção `do... while`, o bloco é executado pelo menos uma vez antes da expressão ser testada.

Exemplo:

```
while (valor <= maxValor) {
 ...
 valor++;    // valor = valor + 1
} // fim do while
```

Iteração for

Sintaxe básica:

```
for (inicial; teste; incremento) {...}
```

`inicial`, `teste` e `incremento` são todas expressões que controlam o loop. Qualquer uma delas ou todas são opcionais.

Exemplo:

```
for (int parte = 10; parte > 1; parte--) {
     fatorial = fatorial * parte;
}
```

Seleção switch

Seleção tipo *switch-case*. É idêntica à expressão `switch` usada em C ou C++. *Sintaxe básica:*

```
switch (expressão) {
```



```
    case constante_1 : expressões; break;
    case constante_2 : expressões; break;
    ...
    case constante_n : expressões; break;
    default: expressões; break;
}
```

Na expressão acima, o `break` é essencial para sair de uma cláusula `case`. A única exceção é se houver um `return` ou ocorrer uma exceção (`Exception`) no código. Se não houver `break` no fim de um dos `cases`, o controle passará ao bloco seguinte e assim por diante até que o `switch` acabe ou um `break` seja encontrado.

Há ainda outras formas de interferir no fluxo de um loop (`for` ou `while`), com `break`, `continue` e rótulos (`labels`): `break` pode ser usada para forçar a saída do loop e `continue` pode ser usado para sair de um loop de vários níveis e recomeçar em um nível mais externo, identificado por um rótulo (um identificador, seguido de ":"). É semelhante ao uso em JavaScript, mas em Java pode-se definir rótulos.

1.4. Vetores

Não existe classe especial para vetores. No entanto, eles também não são tipos primitivos. Vetores (*arrays*) têm, em Java, *status* de objeto, mas não têm uma classe correspondente. Todo vetor possui uma variável pública chamada `length`, que informa o tamanho do mesmo. É o único meio de obter o tamanho de um vetor em Java. No trecho de código abaixo, a variável `x` armazena o valor 4, que corresponde ao comprimento do vetor.

```
int vetor[] = {1, 1, 2, 2};
int x = vetor.length;
```

Além da maneira acima, existem outras formas de declarar e inicializar vetores. A inicialização utiliza a palavra-chave "new" e define a quantidade de elementos do vetor (que devem ter o mesmo tipo), como mostrado nos exemplos abaixo. Também pode-se definir vetores de vetores, que na prática são vetores *multidimensionais*.

Exemplos:

Declaração de um vetor de `Strings` unidimensional;

```
String[] args;
```



```
String args[]; // sintaxe opcional (semelhante a C)
```

Declaração e inicialização de um vetor de `Strings` unidimensional;

```
String[] args = new String[10]; // sintaxe mais clara!
String args[] = new String[10];
```

Declaração de três vetores de inteiros: um tridimensional (`espaço`), um bidimensional (`matriz`) e um unidimensional (`vetor`). Exemplos de sintaxes variadas:

```
int espaço[][][], matriz[][], vetor[];
int[] espaço[][], matriz[], vetor;
int[][] espaço[], matriz; int[] vetor;
int[][][] espaço; int[][] matriz; int[] vetor; // sintaxe mais clara!!!
```

Inicialização de um vetor bidimensional e de um tridimensional:

```
Object[][] square = new Object[15][];
Object[][][] cube = new Object[10][][];
```

Inicialização explícita (só é permitida durante a declaração):

```
int[][] matriz = {{0, 1}, {0, 1, 2}};
```


JAVA 3

Classes e objetos

OS CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS são apresentados neste módulo. Em seguida são demonstrados exemplos de sua implementação na linguagem Java, implementados em exemplos que mostram os conceitos de herança, polimorfismo, abstração e encapsulamento.

Tópicos abordados neste módulo

- Conceitos de programação orientada a objetos
- Classes e objetos em Java
- Abstração, herança, polimorfismo e encapsulamento
- Criação e destruição de objetos
- Pacotes
- Modificadores de acesso e qualidade

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- saber quais são as principais características de linguagens orientadas a objetos
- Saber como se cria uma classe em Java
- Saber como se cria objetos
- Saber criar métodos, variáveis, construtores
- Saber definir métodos abstratos, classes abstratas, interfaces, métodos finais e classes finais.
- Saber criar constantes
- Saber o que são e quando usar Interfaces.

1.1. Programação orientada a objetos

O termo "orientado a objetos" tem sido tão usado e abusado que o seu sentido concreto se perdeu. Tem caracterizado desde modelos de administração a métodos de auto-ajuda. Muitos produtos de software que alegam ser orientados a objetos não o são. Às vezes possuem alguns recursos baseados em objetos, outras vezes nem isso. Toda essa confusão de conceitos fez com que ninguém hoje soubesse mais o que era realmente ser "orientado a objetos".

Java é uma linguagem orientada a objetos e para entender Java é necessário compreender os conceitos da programação orientada a objetos ("POO"). O

objetivo desta seção é esclarecer o significado da POO, apresentando os principais conceitos por trás deste paradigma. Procuraremos usar uma linguagem clara e ilustrar os conceitos através de exemplos do mundo real e em seguida com exemplos de programação. A aplicação desses conceitos na programação em Java será apresentada posteriormente nas seções seguintes.

Apesar de ser um assunto muito badalado, o conceito de POO não é novo. Tem mais de trinta anos de idade: foi introduzido pela primeira vez na linguagem *Simula-67*. Desde então têm surgido várias outras linguagens com esses conceitos como *Smalltalk* e *Eiffel*. C++, a mais popular delas, é na verdade uma linguagem híbrida. É uma extensão do C com suporte a classes e objetos, mas permite que se desenvolva programas sem classes e objetos. *Perl 5* é outra linguagem híbrida. Java, por sua vez, é orientada a objetos desde o princípio. É *impossível* escrever um programa em Java sem usar objetos.

Porque usar objetos?

As vantagens de se usar objetos como blocos para a construção de aplicações são muitas. Podemos citar:

- *Simplicidade:* os objetos escondem a complexidade do código. Pode-se criar uma complexa aplicação gráfica usando botões, janelas, barras de rolamento, etc. sem conhecer complexidade do código utilizado para criá-los.
- *Reutilização de código:* Um objeto, depois de criado, pode ser reutilizado por outras aplicações, ter suas funções estendidas e ser usado como bloco fundamental em sistemas mais complexos.
- *Inclusão Dinâmica:* objetos podem ser incluídos dinamicamente no programa durante a execução. Isso permite que vários programas compartilhem os mesmos objetos e classes, reduzindo o seu tamanho final.

Elementos do modelo orientado a objetos

Os autores divergem quanto às características que fazem uma linguagem ser orientada a objetos, mas a maioria concorda que o paradigma se baseia em quatro princípios básicos: *abstração*, *encapsulamento*, *herança* e *polimorfismo*. Todas as características se relacionam entre si. Explicaremos cada uma delas a seguir.

Abstração

É o processo de extrair as características *essenciais* de um objeto real. A abstração é necessária para se ter um modelo fiel da realidade sobre o qual se possa operar. O conjunto de características resultante da abstração forma um *tipo de dados abstrato* com informações sobre seu estado e comportamento.

A abstração nem sempre produz os mesmos resultados. Depende do *contexto* onde é utilizada. Ou seja, a abstração de um objeto por uma pessoa pode ser diferente na visão de outra. Por exemplo, para um mecânico, um carro é caracterizado pela marca, modelo, potência, cilindradas, etc. Já um usuário comum pode caracterizá-lo pela cor, marca, modelo, consumo e preço. E finalmente, para o *Departamento de Trânsito*, um carro pode ser totalmente caracterizado pela placa, proprietário, marca, modelo, cor e número de chassi.

Vamos supor que você vai escrever um aplicativo para uma transportadora que trabalha exclusivamente para uma livraria e cobra por livro, apesar do custo ser baseado no peso. É possível ter uma boa noção do peso de um livro sabendo o número de páginas, o formato e o tipo de capa (dura ou mole). Portanto, você poderá criar um objeto `Livro` que tenha essas três características. Mas se você também for escrever um aplicativo semelhante para a livraria, não poderá aproveitar o mesmo objeto `Livro` que você criou pois a abstração será diferente. Para a livraria, interessam características como autor, título, assunto, editora, preço, etc. que não interessam à transportadora.

Encapsulamento

É o processo de combinar tipos de dados, dados e funções relacionadas em um único bloco de organização e só permitir o acesso a eles através de métodos determinados. O encapsulamento permite que o usuário do objeto o utilize através de uma *interface pública* bem definida onde todas as funções disponíveis são completamente especificadas.

Por exemplo, existe um número determinado de coisas que se pode fazer com um toca-fitas. Pode-se avançar, voltar, gravar, tocar, parar, interromper e ejetar a fita. Dentro do toca-fitas, porém, há várias outras funções sendo realizadas como acionar o motor, desligá-lo, acionar o cabeçote de gravação, liberar o cabeçote, e outras operações mais complexas. Essas funções são *escondidas* dentro do mecanismo do toca-fitas e não temos acesso a elas diretamente. Quando apertamos o "play" o motor é ligado e o cabeçote de

reprodução acionado, mas não precisamos saber como isto é feito para usar o toca-fitas. A interface pública do toca-fitas, constituída pelos botões, esconde a complexidade do aparelho, tornando o seu uso mais simples e mais seguro.

Uma das principais vantagens do encapsulamento é esconder a complexidade do código. Outra, é proteger os dados. Ao permitir o acesso a eles apenas através de métodos de uma interface pública evita que os dados sejam corrompidos por aplicações externas.

No exemplo do toca-fitas, quando usamos o "método" *gravar*, sabemos que o aparelho irá gravar informações na fita em um formato padrão que poderá ser lido por outros aparelhos similares. Se não existisse o método *gravar* e tivéssemos acesso direto à fita, cada pessoa poderia estabelecer uma maneira diferente de gravar informações (usando laser, imãs, etc.). Gravar uma fita passaria a ser uma tarefa complicada, pois o usuário teria que conhecer toda a complexidade do aparelho. Depois, se perderia a compatibilidade pois não haveria uma forma de proteger os dados para que fossem sempre gravados em um formato padrão. O encapsulamento, então, funciona tanto para proteger os dados como para simplificar o uso de um objeto.

Há vários exemplos na programação. Os aplicativos que conhecemos e que rodam no Windows tem uma grande semelhança porque usam os mesmos objetos. Cada objeto tem um número limitado de funções (métodos) que podem ser invocados sobre ele. Se você decide usar, em alguma parte do seu programa, dois botões: **OK** e **Cancela**, você não precisa entender como um botão foi escrito, nem saber quais as suas variáveis internas. Basta saber como construí-lo, mudar o texto que contém e depois incluí-lo em seu programa. Não há como corromper a estrutura interna do botão.

Herança

É o aproveitamento e extensão das características de uma classe existente.

A melhor analogia neste caso é a natureza. No reino animal, os mamíferos herdam a característica de terem uma espinha dorsal por serem uma subclasse dos vertebrados. Acrescentam a essa característica, várias outras, como ter sangue quente, amamentar, etc. Os roedores herdam todas as características dos vertebrados e mamíferos e acrescentam outras, e assim por diante.

Em programação, a herança ocorre quando um objeto aproveita a implementação (estruturas de dados e métodos) de um outro tipo de objeto para desenvolver uma especialização dele. *Tipo de objeto* é um termo que já usamos antes. Refere-se a uma característica comum de um grupo ou classe de objetos. Em POO, tipos de objetos são definidos por um protótipo chamado de "classe". A herança permite a formação de uma hierarquia de classes, onde cada nível é uma especialização do nível anterior, que é mais genérico.

É comum desenvolver estruturas genéricas para depois ir acrescentando detalhes em POO. Isto simplifica o código e garante uma organização maior a um projeto. Também favorece a reutilização de código. Se precisamos implementar um botão cor-de-rosa e já temos uma classe que define as características de um botão cinza, podemos fazer com que nossa implementação seja uma "subclasse" de `Botão_Cinza`, e depois estender suas características. Desta forma, nos concentramos apenas no necessário (a cor do botão) e evitamos ter que definir tudo a partir do zero.

Polimorfismo

É a propriedade de se utilizar um mesmo nome ou forma para fazer coisas diferentes.

Por exemplo, a ordem: "vire à esquerda!" pode produzir resultados diferentes dependendo da situação. Se quem recebe a ordem estiver dirigindo um automóvel, irá mover a direção para a esquerda; se estiver controlando o leme de uma lancha, moverá a cana do leme para a direita.

Em programação, uma mesma mensagem poderá provocar um resultado diferente, dependendo dos argumentos que foram passados e do objeto que o receberá. Por exemplo: o envio de uma instrução "`desenhe`" para uma *sub*classe de "`polígono`", poderá desenhar um triângulo, retângulo, pentágono, etc. dependendo do objeto que receber a instrução.

Isto é muito útil para escrever programas versáteis, que possam lidar com vários tipos diferentes de objetos. Por exemplo, pode-se projetar um `Posto` para abastecer qualquer tipo de transporte. Na classe `Transporte`, poderíamos definir um método "abastece" genérico. Suponha agora que `CarroElétrico`, `Jegue`, `Porsche` e `AirBus` sejam subclasses de `Transporte`. Cada um tem sua própria maneira de se abastecer (bateria, ração, gasolina ou querosene), mas a única coisa

que o `Posto` precisa fazer, é acionar o método "`abastece`" em cada um deles. Não precisa saber como é feito. Os detalhes de cada procedimento ficam com o objeto.

O tipo de polimorfismo do exemplo acima chama-se *sobreposição de métodos*, pois cada subclasse *sobrepôs* o método abastece definido na classe `Transporte`, e o redefiniu com uma nova implementação.

A vantagem desse tipo de polimorfismo em programação é a possibilidade de desenvolver programas genéricos que utilizam uma interface padrão. Por exemplo, criar um programa que opere sobre objetos do tipo `Transporte` que possuam um método abastece. Mesmo que um novo tipo de transporte seja inventado após a criação do programa, se este novo tipo estender a classe `Transporte` ele será compatível com o nosso programa. Em Java, inclusive, é possível implementar polimorfismo sem herança através de classes especiais chamadas de *interfaces*. Uma interface não tem métodos implementados, apenas suas assinaturas (tipo de retorno, nome e argumentos). Cada classe que a estende é obrigada a implementar cada método nela definido.

Outro tipo de polimorfismo é o que se chama de *sobrecarga de nomes*. Na sobrecarga usa-se o mesmo nome e mais alguma característica para se fazer a mesma coisa.

Suponha que existe um carro `Híbrido` que usa gasolina e eletricidade. Quando ele chega no posto, o seu método `abastece()` é chamado e ele enche as suas duas fontes de energia. Suponha que ele pode também ter esse método invocado pelo `Posto` com argumentos diferentes como `abastece(`*`qual_fonte`*`)`. Então se o posto aciona o método `abastece(gasolina)` em `Híbrido`, ele preenche somente o seu tanque de gasolina e não recarrega as baterias. Então, dependendo da chamada `abastece()` ter ou não argumentos, o abastecimento também poderá ocorrer de forma diferente.

O que é um objeto?

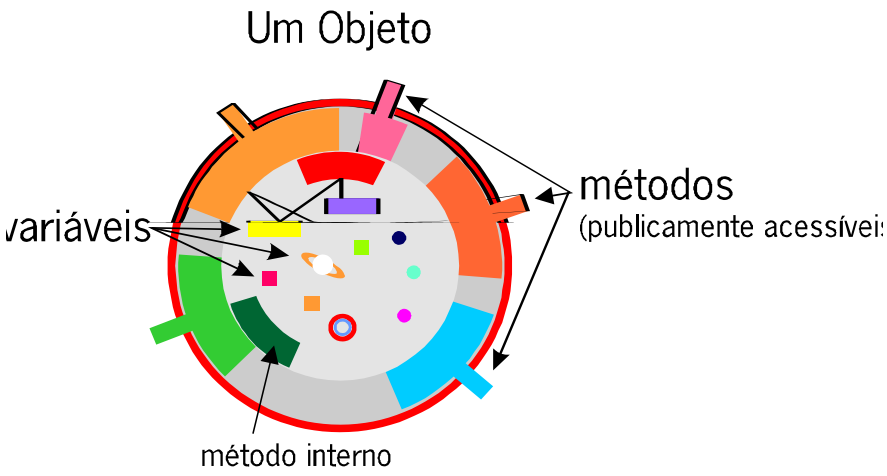
Objetos são a base da tecnologia orientada a objetos. Consistem de modelos (abstrações) de objetos reais[1]. Eles preservam as características essenciais de um objeto real: o seu *estado* e o seu *comportamento*.

[1] Objetos também podem ser usados para modelar conceitos abstratos, como eventos. Um evento é um objeto usado em sistemas de janela para representar as ações do usuário (clíques do mouse , teclas digitadas).

Qualquer objeto real pode ser abstraído para filtrar seu estado e comportamento. Vejamos alguns exemplos:

OBJETO	ESTADO	COMPORTAMENTO
Carro	*velocidade atual, marcha, cor, modelo, marca*	*troca marchas, acelera, dá partida, freia*
Gato	*nome, raça, com fome, com preguiça*	*mia, dorme, se estica, brinca, caça, liga o "motor"*
Caneta	*cor, nível de tinta*	*escreve, coloca mais tinta*
Toca-fitas	*ligado, girando, sentido da fita, tocando, gravando*	*grava, toca, avança, volta, para, pausa, ejeta, liga*

Em programação orientada a objetos, o estado de um objeto é representado por *campos de dados* (variáveis), e seu comportamento é implementado com *métodos* (funções). A única forma de mudar o estado dos dados é através de métodos[2]. Por exemplo, para mudar o estado do toca-fitas para ligado, é necessário utilizar o método "liga". A figura abaixo representa um objeto de software:

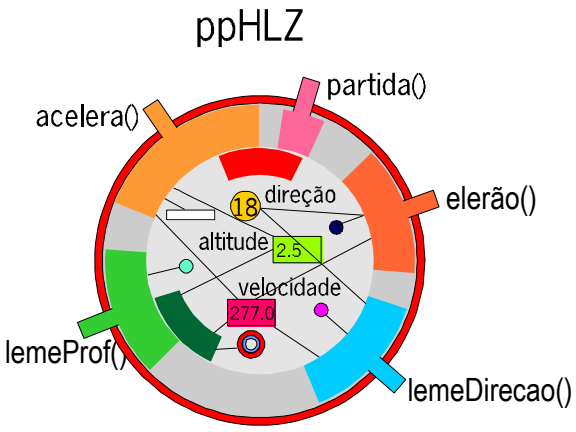


Tudo o que o objeto de software sabe (seu estado) e tudo o que ele pode fazer (o seu comportamento) é determinado pelos seus campos de dados e pelos seus métodos.

Vamos representar o modelo de um avião *Cessna 182*, através de um objeto de software. As variáveis indicam o seu estado atual: `velocidade = 277km/h`, `direção = 18`, `altitude = 2,6km`. `Partida()`, `acelera()`, `lemeDirecao(`*`direção`*`)`, `elerao(`*`direção`*`)`, `lemeProfund(`*`posição`*`)` e `altímetro()` são métodos que podem mudar o estado das variáveis fazendo o

[2] A maioria das linguagens (inclusive Java) permite a declaração de variáveis públicas, que não precisam respeitar ao encapsulamento.

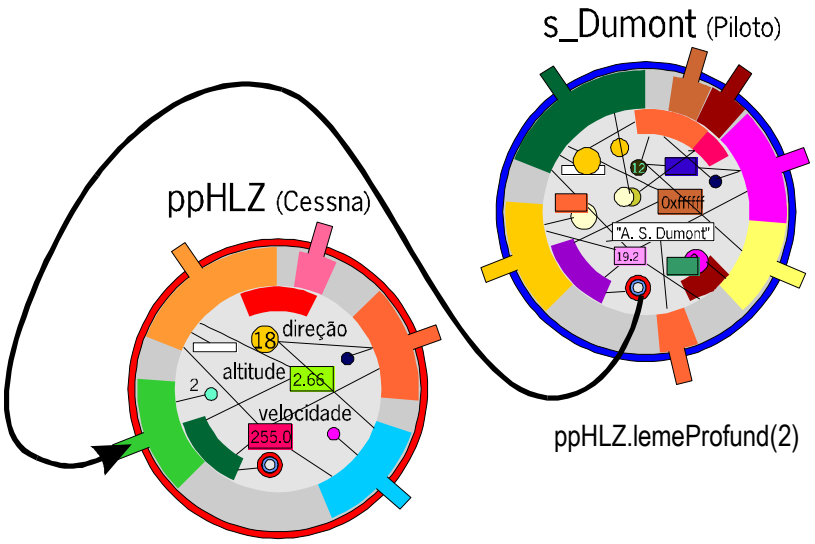
avião ganhar altitude, velocidade ou mudar de direção ou fornecer informações importantes. O diagrama a seguir ilustra o objeto, que chamamos de `ppHLZ`:



Um objeto sozinho não tem muita utilidade. Um avião sozinho no hangar e sem piloto só serve de enfeite. Para se conseguir alguma funcionalidade dos objetos, é necessário que haja uma interação com outros objetos. Só assim, se pode construir modelos de comportamento mais complexo.

Objetos de software se comunicam entre si através de mensagens. Como não temos acesso às variáveis a não ser através de métodos, a troca de mensagens consiste da invocação de métodos e o possível retorno de valores. No exemplo do `Cessna`, é necessário que haja um outro objeto, um `Piloto`, que envie mensagens para o avião para que ele possa sair do chão.

A figura abaixo ilustra esta situação. O avião voa para o Sul (180º) a 277km/h e 2,5km de altura. O objeto `s_Dumont` (um certo piloto) invoca o método `elevador()` sobre `ppHLZ` com o parâmetro `+2`. O método é então executado, provocando o movimento do elevador para cima duas posições, aumentando a altitude e reduzindo a velocidade (mudando o estado das variáveis de `ppHLZ`).



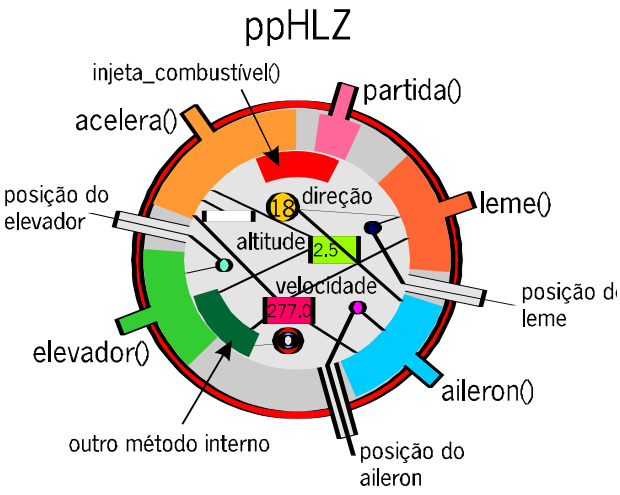
Para realizar a mudança acima em Java, dentro do código do objeto `s_Dumont` haveria uma linha do tipo:

```
ppHLZ.lemeProfund(+2)
```

É assim que Java representa a chamada do método de um objeto. O primeiro nome (`ppHLZ`) é o objeto. Após o ponto, vem o nome de um método com seus atributos (se existirem). No exemplo acima, o método é `lemeProfund()`. Todos os métodos em Java consistem de um nome seguido de `()`, mesmo que não aceitem parâmetros.

Métodos internos e variáveis públicas

Observe que a única maneira de se mudar qualquer estado do avião é através dos seus *métodos públicos*. Os campos de dados são privativos e o seu acesso direto não é permitido ao mundo externo. Se os campos de dados fossem públicos, poderíamos alterá-los diretamente, comprometendo o funcionamento normal do objeto. A figura a seguir ilustra uma versão mais complexa e menos segura do `Cessna ppHLZ`, onde é possível alterar os valores das posições do leme de direção, elerões e leme de profundidade sem usar métodos.



Isto é como mover os lemes e elerões com a mão em vez de usar o manche e o pedal! Torna o avião mais difícil de usar e compromete a sua segurança. A maioria das linguagens de programação, inclusive Java, permitem que se use campos de dados públicos. Geralmente as desvantagens desse procedimento são maiores que as vantagens, portanto, deve-se usar o encapsulamento dos dados pelos métodos o quanto possível.

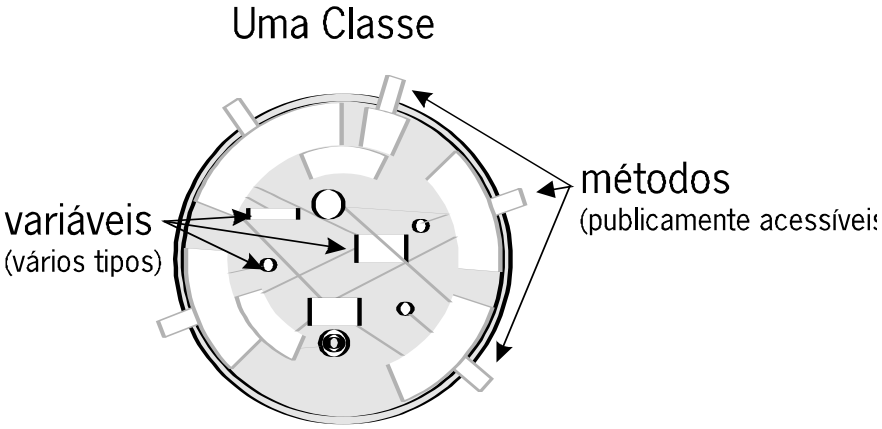
Um objeto complexo poderá ter centenas de métodos, mas talvez somente uns quatro ou cinco interessem a quem vai usá-lo. Quem dirige um carro não precisa saber de coisas como injetar combustível, provocar faísca, etc. Simplesmente ele liga a ignição. Neste caso, ligar a ignição é um método público, que interessa ao motorista. Já injetar combustível, provocar faísca, são métodos internos (ou privados) ao funcionamento do automóvel, que podem ser chamados por outros métodos sem que o motorista precise se preocupar.

O que são classes?

Classes são protótipos utilizados para construir objetos. As características de estado e comportamento do objeto `ppHLZ`, ilustrado nos exemplos anteriores, são características de qualquer avião Cessna 182RG que sai da fábrica. Estas características são definidas nas classes, que podem ser vistas como a *planta* usada para construir o avião.

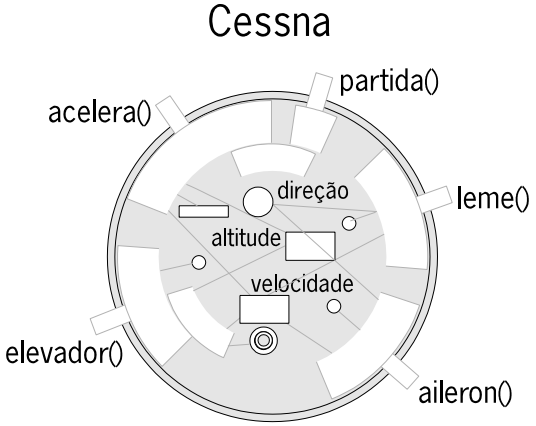
A figura abaixo representa uma classe. Observe a semelhança com o diagrama que utilizamos para representar os objetos. A classe não é um objeto pois não possui um estado, nem um comportamento específico. Ela não é algo

que existe e que possa ter estados. Por isso, aparece no diagrama como um fantasma, sem cores, pois representa apenas um gabarito com o qual podem ser construídos objetos.



Com as plantas que contém o projeto de um Cessna, pode-se construir vários aviões Cessna. O projeto inclui as especificações do motor, da fuselagem, da asa, vários desenhos de perspectiva, moldes e até uma maquete em miniatura. Descreve em detalhes tudo o que o Cessna pode fazer. Como vão funcionar os lemes, os elerões, o trem de pouso, etc. Contém tudo o que um Cessna tem mas não é um Cessna.

Para produzir os aviões `ppHLZ`, `ptANX` e `ptGRN`, todos do modelo Cessna 182, definimos uma classe chamada `Cessna`. Esta classe possui a declaração das variáveis que serão utilizadas para armazenar a direção, velocidade e altitude, além da implementação dos métodos que cada Cessna deve ter.



Os objetos são chamados de *instâncias* das classes. Quando criamos um objeto, dizemos que instanciamos uma classe. Cada instância é uma entidade individual e pode então receber valores para as suas variáveis e modificá-las através de métodos. Com uma planta, fazemos 100 aviões, que podem então ter velocidades, altitudes e direções diferentes.

Tendo criado a classe `Cessna`, podemos criar agora vários aviões com as mesmas características básicas. Por exemplo, se `ppHLZ`, `ptANX` e `ptGRN` são instâncias de `Cessna`, todos têm uma velocidade, uma altitude e uma direção distintas e podem dar partida, acelerar, subir, descer, etc. `ppHLZ` pode estar a 350km/h voando para o Norte, enquanto `ptANX` pode estar estacionado no hangar.

A grande vantagem de se usar classes é a reutilização do código. Toda a estrutura de um objeto é definida na classe, que depois é utilizada quantas vezes for necessário para criar vários objetos. O próprio avião pode ser construído a partir de objetos menores. Por exemplo, os objetos `tanque`, `hélice`, `motor`, `roda` podem ser peças que a fábrica de aviões já compra pronta de outra fábrica. A fábrica de aviões não precisa saber como faz uma hélice e nem precisa ter o molde ou planta que explica como fazê-la. Simplesmente usa o objeto. Já uma fábrica de hélices, possui uma classe `Hélice`, conhece os detalhes de uma hélice, faz centenas delas e as vende para fábricas de aviões.

Com orientação a objetos, o programador se preocupa com o objetivo principal do programa que quer desenvolver. Não fica preocupado com detalhes como programação de botões, molduras, interfaces gráficas, hélices, etc. Usa objetos pré-fabricados para montar a infraestrutura do seu projeto e se concentra na implementação específica do problema que pretende resolver.

Herança

Não bastasse a reutilização do código proporcionado pelas classes, na criação de objetos, a POO vai mais longe e define meios de permitir em uma classe, a reutilização do código de outras classes.

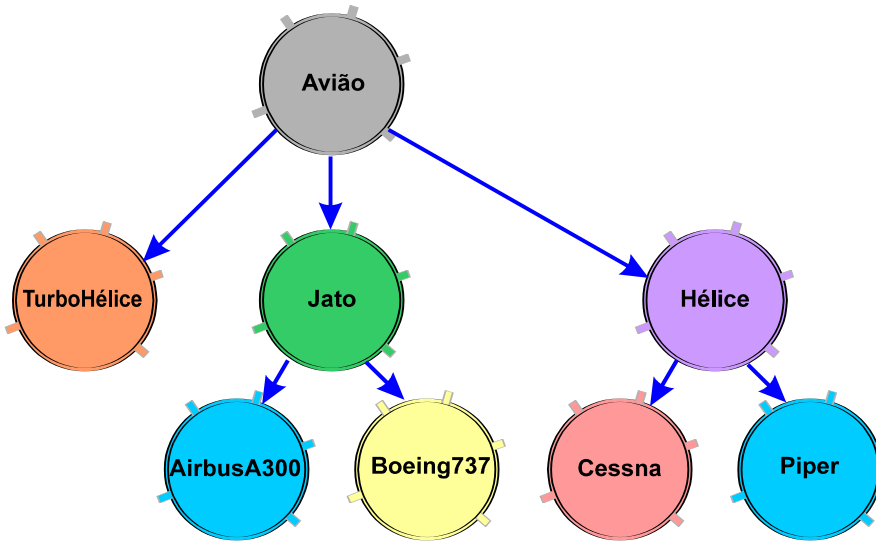
As características cor, velocidade, marcha, etc. podem estar definidas em uma classe chamada de `Avião`, que declara métodos `elevador()`, `partida()`, `leme()`, etc. Esses métodos e variáveis podem ser herdadas pelas classes `AHélice` e `AJato` que são tipos de aviões. A classe `Cessna` pode ser uma subclasse de

`AHélice` e a classe `AJato` pode ter outras subclasses como `F15`, `Boeing` e assim por diante.

Em cada classe, métodos novos podem ser criados, outros podem ser redefinidos e novos campos de dados podem ser declarados. Cada subclasse adiciona alguma nova funcionalidade à classe pai e continua sendo um objeto do tipo definido pela raiz das classes. Ou seja, tanto `Cessna`, `AirbusA300`, e `Boeing737` são aviões. `Cessna` e `Piper` são aviões e são movidos a hélice.

É importante observar esta relação de "ser" no projeto de sistemas orientados a objetos. Por exemplo, pode-se criar uma classe `Esfera` que estende `Círculo` e redefine alguns de seus métodos. `Esfera` e `Círculo` têm várias semelhanças, mas uma esfera *não é* um círculo e poderá haver problemas em uma fase posterior da implementação quando tentarmos passar uma esfera para um objeto que lida com formas bi-dimensionais.

A figura abaixo ilustra uma hipotética hierarquia de classes formada pelas subclasses da classe `Avião`.



Todos os métodos e dados públicos de uma superclasse são herdados pelas suas subclasses. Desta forma, se `Avião` já define o método `partida()`, não há necessidade de `Cessna` redefini-lo. Os objetos criados com a classe `Cessna` já nascem com um método `partida()`, mesmo que este não faça parte da planta do avião.

Mas, na maioria das vezes, a maneira como uma classe genérica define uma ação não serve para classes mais específicas. Provavelmente a partida na classe

`Avião` é manual. Neste caso, a subclasse pode sobrepor ou sobrecarregar os métodos herdados. O `Cessna`, por exemplo, pode ou definir um novo método `partida(tipo)` que suporte tanto partida manual como elétrica, mantendo o método `partida()` para acionamento manual (*sobrecarregando* o método), ou simplesmente rescrever `partida()` para que faça somente partida elétrica (*sobrepondo* o método).

`Cessna`, por exemplo, pode acrescentar novos métodos como `baixarTrem()` e `horizonte()` (para ler as indicações do horizonte artificial). `Jato` deve sobrepor o métodos `partida()`, `elevador()`, `leme()` com uma implementação específica para um jato genérico. Talvez acrescente `flaps()`, `pilotoAut()`. `Boeing737` também pode acrescentar novas variáveis `capacidadeDeCarga`, `númeroDePassageiros`, etc. para caracterizar melhor os objetos que serão produzidos.

Exercícios

Nas seções seguintes utilizaremos exaustivamente os conceitos aqui apresentados que são básicos para a linguagem Java, portanto a sua compreensão é indispensável. Os exercícios a seguir ajudarão a fixá-los.

1. Faça a representação (estado e comportamento) de um objeto do tipo FerrariF40. Liste alguns métodos e campos de dados e represente-o através de um diagrama.
2. Mostre como um objeto do tipo Motorista poderia invocar alguns dos métodos de FerrariF40 para alterar o seu estado.
3. Como poderíamos construir uma FerrariF40? Liste alguns objetos que poderiam ser utilizados como parte de uma Ferrari como rodas, bancos, motor, freios. Represente-os esquematicamente e liste os seus métodos. Mostre como a Ferrari poderia usar os seus componentes (como a Ferrari poderia invocar métodos de seus componentes. Ex: motorista aciona ferrari.partida() que aciona motor.partida())
4. Construa uma hierarquia de classes para Veículos Motorizados, que poderia ter como uma de suas subclasses, a classe Automóvel. Represente as outras classes e encaixe FerrariF40 em algum lugar da hierarquia. Quais métodos (e campos) poderiam ser definidos na classe raiz da hierarquia. Quais deveriam ser sobrepostos.

5. Represente uma classe HondaTitan125 na hierarquia de classes do exercício anterior. Quais os métodos que deveriam ser sobrepostos (ou definidos) para motos? Em que classes da hierarquia devem ser sobrepostos cada método?
6. Identifique os casos onde está havendo abuso da reutilização do código, nas subclasses que estendem as superclasses abaixo: 1) Esfera estende Circulo; 2) Círculo estende Elipse; 3) Globo estende Esfera; 4) TV estende RádioFM; 5) TVsemImagem estende TV; 6) Homem estende Macaco; 7) Homem estende Primata. (Dica: verifique se a subclasse "é" um objeto do tipo da superclasse ou se apenas "parte" dele está na superclasse).

1.2. Classes e objetos em Java

Como vimos na seção anterior, uma classe é um *gabarito* que define os dados e métodos comuns a todos os objetos do seu tipo. Desta forma, a classe caracteriza uma qualidade, ou *tipo* de um objeto.

A classe, em Java, é um *tipo de dados abstrato* que pode ser definido pelo usuário. É, na verdade, algo mais que isto. Uma classe consiste de declarações de *campos de dados* e *métodos*. Os campos de dados são variáveis, referências ou constantes e os métodos contém o código com as instruções que operam sobre os dados da classe ou dos objetos criados através dela. Esses dois ingredientes, campos de dados e métodos, permitem a criação de objetos com um certo *estado* (descrito nas variáveis) e *comportamento* (definido pelos métodos).

A forma mais simples de se criar uma classe em Java é usando a palavra-chave `class` seguida por um identificador que serve de nome para a classe. O corpo da classe é totalmente delimitado por chaves, e contém as declarações dos campos de dados e métodos. A ordem dos fatores aqui não é importante.

```
class Nome_da_classe {
 declaração/definição de campos (...)
 declaração/definição de métodos (...)
}
```

Por exemplo, a classe `Cessna`, que representamos esquematicamente no capítulo 2, poderia ser definida inicialmente da seguinte forma:

```
class Cessna {
 private int direção;
 private float altitude;
 private float velocidade;
```



```
    public void lemeProfund(int posição) {
        // instruções para mudar
        // posição do elevador
    }

    public void lemeDirecao(int posição) {
        // instruções para mudar
        // posição do leme
    }

    public boolean partida() {
        // instruções para dar partida
           return true; // partida foi dada
    }

    public float altimetro() {
           return altitude;
    }
}
```

Nesta definição há três variáveis de *instância* (variáveis declaradas no corpo da classe que não são `static` só podem ser usadas por objetos, e são chamadas *de instância*) e quatro métodos. As variáveis são privativas à classe, ou seja, não são acessíveis quando se utiliza um objeto do tipo `Cessna`. A única forma de se obter um registro da altitude do avião, por exemplo, é através da invocação do método `altímetro()`, que, como os outros métodos desta classe, é publicamente acessível.

Observe que, neste modelo, não é possível para um outro objeto (do tipo `Piloto`, por exemplo) saber a velocidade do avião criado com esta classe, pois não criamos um método `velocímetro()` que retornaria o valor da variável `velocidade`. Só são visíveis fora da classe os quatro métodos.

É ainda possível usar muitos outros métodos não definidos nesta classe como `equals()`, `clone()`, etc. De onde eles vêm? Da superclasse `Object`! Veremos mais sobre extensão de classes no capítulo seguinte, mas é importante lembrar (do capítulo 3) que toda classe em Java é a extensão de uma classe de nível mais alto na hierarquia. Se esta classe não é especificada (como nos nossos exemplos até agora), ela é subclasse da classe `Object`, que ocupa o nível mais alto da hierarquia e faz parte da biblioteca fundamental da linguagem Java (não precisa ser importada).

Variáveis

Toda declaração de variável obrigatoriamente deve informar o seu tipo. No exemplo acima, temos uma variável do tipo `int` e duas do tipo `float`. Os métodos, por sua vez, devem informar o *tipo que retornam* e caso não retornem valor algum devem ser declarados como `void`. É o caso do método `leme()`, no exemplo acima, que só trata de mudar a posição do leme e não retorna informação alguma.

A sintaxe geral para a declaração de variáveis é a seguinte:

```
[modificadores] tipo nome_da variável;
```

Os modificadores são opcionais. Podem modificar a visibilidade (acesso) de uma variável ou a sua qualidade. Os modificadores de visibilidade são: `public`, `private` e `protected`, ou a ausência de um deles. Esses modificadores definem que outras classes e objetos podem acessar a variável. As variáveis do nosso exemplo foram declaradas `private`, portanto não são visíveis de fora do objeto, a não ser que, como fizemos com a variável altitude, o seu valor seja repassado através de um método público.

Os modificadores de qualidade para variáveis são:

- **`final`**: define uma constante. Na declaração de uma constante, o seu valor já deve ser definido. Por exemplo, poderíamos ter em um programa que calcula órbitas de satélite, a gravidade da Terra ao nível do mar em m/s:

  ```
  public final g = 9.810;
  ```

- **`static`**: usado para declarar variáveis estáticas ou globais que são acessíveis somente através da classe e independente da criação de objetos. As variáveis estáticas são alocadas na hora em que a classe é carregada. Ocupa uma dimensão a parte da dimensão dos objetos. Se uma variável é declarada estática, só há uma cópia dela independente do número de objetos criados. Cada novo objeto criado tem acesso a esta variável e ela pode ser usada como forma de definir valores globais visíveis entre objetos.

- **`transient`:** usado em serialização de dados para informar que a variável não deve ser salva como parte integrante do objeto. Não vale para campos estáticos.

- **`volatile`:** usado para informar o compilador que ele não deve tentar fazer otimizações com esta variável. Vale para campos usados em linhas de execução (*threads*) sincronizadas.

Quanto aos modificadores de qualidade, no momento só nos interessam dois (os outros veremos mais na frente):

- **`public`:** torna a variável acessível de fora do objeto, ou seja, qualquer outro programa pode mudar o seu valor usando a notação:

```
objeto.variável
```

- **`private`:** torna a variável inacessível de fora da classe ou do objeto.

Na grande maioria dos casos, um projeto não deve ter variáveis públicas. Sempre que possível, o estado de um objeto só deve ser modificável através de um método. Isto garante a integridade dos dados evita a corrupção de dados ou o mau uso de um objeto. Por exemplo, suponha que alguém criasse um círculo e alterasse seu raio:

```
Círculo c1 = new Círculo();
c1.raio = -5;
```

Não existem círculos com raios negativos. O programa que desenha o mesmo iria falhar por causa disto. Uma forma de evitar o problema é declarar raio como `private` e fornecer um método para alterar o seu valor e outro para permitir a leitura:

```
public double raio() {
   return raio;
}

public void mudaRaio(double novoRaio) {
   if (novoRaio != 0.0) {
       raio = Math.abs(novoRaio);
   } else {
       raio = 1.0;
   }
}
```

Esta mudança fará com que a única forma de mudar o raio seja através de um método:


```
c1.mudaRaio(-5);
```

O método fará com que e o sinal seja sempre ignorado e que raios nulos tenham pelo menos o valor 1.0, protegendo o objeto.

Métodos

A sintaxe geral para a declaração de métodos é a seguinte:

```
[modificadores] tipo_de_retorno nome_do_método ([tipo argumento,
tipo argumento, ...]) [throws Exceção] { ... }
```

O que está entre `[` e `]` é opcional. Os modificadores podem modificar a visibilidade (acesso) de um método ou a sua qualidade. Os modificadores de visibilidade novamente são: `public`, `private` e `protected`, que definem que outras classes e objetos podem acessar o método. Todos os métodos do nosso exemplo são públicos, por isso são acessíveis de fora.

Veremos os modificadores de visibilidade (ou acesso) em mais detalhes na próxima seção.

Os modificadores de qualidade para métodos são:

- **`final`:** faz com que a implementação do método seja uma versão final e não possa ser sobreposto em uma sub-classe, ou seja, se definíssemos `Cessna` como sub-classe de `Monomotor`, e tivesse um método `altímetro()` definido nesta classe como final, *não* poderíamos redefiní-lo em `Cessna`.

- **`abstract`:** usado para definir um método sem implementação, que necessariamente precisa ser *sobreposto* em uma sub-classe para que possa ser usado. Por exemplo, declarando o método `partida()` como `abstract` em `Avião`, tornará sua implementação obrigatória nas subclasses (todo avião deve ter um meio de dar partida) de uma forma particular a cada caso.

- **`native`:** declara um método que usa uma implementação escrita em outra linguagem (C ou C++, por exemplo).

- **`static`:** usado para declarar um método como pertencente à classe. Este método funciona mais como uma função pois não opera em cima de um objeto. Pode ser chamado através da própria classe (sem precisar

criar um objeto antes). Assim como as variáveis estáticas, os métodos estáticos estão em uma dimensão diferente daquela dos objetos. Eles agem como se estivessem completamente separados da classe (e podem ser separados sem prejudicar os objetos). Métodos estáticos não podem usar campos de dados ou métodos não-estáticos. O trecho seguinte *não* compila:

```
1. public class Thing {
2.
3.    private int size;
4.
5.    public static void main(String[] args) {
6.       size = 10;          // illegal
7.       System.out.println("Size is " + size);
8.    }
9. }
```

- **`synchronized`:** em programas que usam *multithreading*, este modificador garante que os dados manipulados pelo método não serão modificados por outro método enquanto este não terminar de usá-los.

A cláusula `throws` é usada quando um método causa uma exceção que não é capturada e tratada dentro do mesmo. É necessária esta declaração para que os programas que usem o método possam tentar capturar a exceção ou passá-la adiante.

O nome do método deve sempre ser seguido de parênteses, mesmo se não houver argumentos. Se houver, o tipo deles deve ser declarado. Os argumentos são separados por vírgulas. Toda a implementação do método deve vir entre as chaves { e } a menos que seja declarado `abstract` ou `native`. Neste caso, a declaração deve terminar com um ponto-e-vírgula:

```
public abstract boolean partida();

private native double integral(float expr, double sup, double inf);
```

Exercícios

1. Implemente um método `velocímetro()`, para `Cessna`, que leia a variável `velocidade` e um método `bússola()` que leia a variável `direção`.
2. Crie uma classe `Círculo` que tenha as variáveis `x`, `y` (coordenadas do centro) e `raio`. Todas devem ser públicas e de ponto-flutuante de simples precisão (`float`). Implemente 2 métodos para o `Círculo`: `area()` e

`circunferência()`. Ambos são públicos e devem retornar os valores como `float`. *Fórmulas do círculo*: Área = π * *raio*^{2}; Circunferência = 2 * π * *raio*.

3. Quais as vantagens e desvantagens de se ter variáveis públicas, como as do exercício 2?
4. Como você escreveria os seguintes tipo de dados abstratos em Java?

   a) Uma casa é uma construção que tem um número, um nome de rua e um proprietário.

   b) Um gato é um animal de estimação que tem um nome, um humano que é seu dono e uma idade.

   c) Uma família tem uma casa, um conjunto de humanos e um animal de estimação.

Criação de Objetos

Objetos são instâncias de uma classe. Não se pode fazer nada com uma classe pois ela só é uma representação genérica de um objeto. A classe é algo imaterial e eterna no "mundo dos objetos", que são "mortais". Os objetos são criados à imagem das classes, mas não podem vê-las. Vivem em outra dimensão. Para podermos usar uma classe como `Cessna` e `Círculo`, precisamos criar instâncias delas, definindo uma determinada aeronave Cessna ou um círculo específico.

Como definimos as classes `Cessna` e `Círculo` (exercício 2) elas agora são tipos em Java, podemos declarar outras variáveis com esse tipo:

```
Cessna ppHLX;
Cessna ptDAT;
Círculo c1;
```

Mas ainda as variáveis `ppHLX`, `ptDAC` e `c1` são apenas nomes que servem de *referência* a objetos `Cessna` ou `Círculo`. Não foi alocada ainda memória na criação de um objeto. Para efetivamente *criar* o objeto, precisamos usar a palavra-chave `new`, chamar uma função que construa o objeto (o construtor) e atribuir o resultado dessa operação à variável declarada como referência:

```
Cessna ppHLX;
Cessna ptDAT;
Círculo c1;
```



```
ppHLX = new Cessna();
ptDAT = new Cessna("Particular");
c1 = new Círculo();
```

Com esta operação, criamos um objeto `Círculo` e dois objetos `Cessna` e os atribuímos às variáveis `c`, que é do tipo `Círculo` e `ppHLX` e `ptDAT`, que são do tipo `Cessna`, respectivamente.

Observe que após o `new`, existe a chamada à um *método* (parece um método pois vem seguido de um par de parênteses), que tem o mesmo nome que a classe. Este é o *construtor* que define como é construído o objeto. A rigor, um construtor não é um método pois não pode ser chamado em outra ocasião a não ser aquela em que constrói um objeto. Depois, o único valor que retorna é uma referência à instância de um objeto. Construtores não são herdados e em alguns casos podem ser chamados mesmo não aparecem definidos na classe O construtor pode ser definido como uma função de inicialização de objetos.

O construtor sempre tem que ser invocado, mesmo que a classe não o defina. Ele é quem monta a infraestrutura básica do objeto. Como toda classe Java estende outra, o construtor sempre chama um outro construtor definido na sua superclasse. Por isso, quando não são usadas construções complexas, pode-se omitir o construtor já que ele sempre chama o da superclasse. Na nossa classe `Círculo` e na classe `Cessna`, o programa chama o construtor de `Object` na hora em que o seu construtor (invisível) é chamado.

O construtor que usamos é básico. Chama-se construtor *default* e não recebe parâmetros. Ele é fornecido *gratuitamente* pelo sistema. Algumas classes podem definir construtores que recebem parâmetros. Assim, a construção do objeto poderia definir previamente alguns valores (como mostra a criação do segundo `Cessna` acima). Neste caso, o construtor precisa estar definido na classe.

Agora que criamos os objetos, podemos acessar suas variáveis e métodos dentro do nosso programa (pode ser a mesma ou outra classe). Fazemos isto usando um ponto "`.`". Por exemplo, no programa do Piloto, ele pode ter uma instrução que invoca um método do avião:

```
ptDAT.lemeProfund(-2); // move o leme de profundidade para
                       // baixo em 2 posições
```

Suponha que também foi implementado no avião Cessna também um método:


```
public void acelera(int quantidade) {
    (...)
}
```

onde cada chamada a este método cause o aumento da aceleração do avião (e como conseqüência aumente a velocidade). Uma simples decolagem poderia ocorrer da seguinte forma:

```
class Piloto {
    public static void main(String[] args) {
        // criação do objeto
        Cessna meuAvião = new Cessna();

        // decolagem
        meuAvião.partida();
        meuAvião.acelera(100);
        if (meuAvião.velocímetro() < 70) {
            meuAvião.lemeProfund(+2);
        }
        if (meuAvião.altimetro() > 500) {
            meuAvião.acelera(-20);
        }
    }
}
```

No exemplo do círculo, podemos manipular com as variáveis pois elas são públicas. Sendo assim, podemos definir as coordenadas e o raio do círculo diretamente:

```
c1.x = 2.0f;  // centro em (2,2) e raio 1.
c.y = 2.0f;
c.raio = 1.0f;
```

Agora podemos fazer qualquer coisa que os métodos e variáveis dos objetos permitirem, como por exemplo, calcular a área do círculo:

```
Círculo c2;
c2 = new Círculo();
float a;
c1.raio = 5;
a = c1.area();     // área retorna um valor float
```

Observe que os métodos e variáveis operam em uma instância específica de `Círculo`. Não precisamos passar argumento para `area()` porque sabemos que o objeto sobre o qual operamos é `c1`. É implícito na linguagem que o método opera nas instâncias da classe na qual é definido.

Vejamos uma possível definição para `area()`:

```
public float area() {
    float a = 3.14159 * raio * raio; // a é variável
    return a;                        // local
}
```

Como é que um método que não recebe argumentos sabe em que dados operar? Na verdade, ele sabe. O argumento está implícito. É o objeto através do qual o método é invocado. Freqüentemente, esta relação é reforçada através da palavra-chave `this`. `this` se refere a "este" objeto, ou seja, a variável `r` utilizada pelo método `public` refere-se à variável `raio` do objeto tipo `Círculo` atual. Quando não aparece explicitamente, é porque geralmente não é necessária, como no exemplo abaixo, e é incluída apenas por clareza. O método `area()` pode ser rescrito da seguinte forma:

```
public float area() {
    return (3.14159 * this.r * this.r);
}
```

O uso de `this`, mesmo quando não obrigatório, ajuda a tornar o código mais claro.

Construtores

Quando criamos um objeto, chamamos uma função que possui o mesmo nome que a classe usada para criá-lo. Esta função especial, como vimos, é chamada de *construtor*.

Todo objeto têm um construtor e ele tem o mesmo nome que a sua classe. Pode ser definido explicitamente, mas se não for, o sistema fornece um construtor *default*. Listamos a seguir, uma implementação da classe `Círculo`, (proposta no exercício 2). Incluímos um construtor para que ele inicialize cada objeto criado já com um valor para o centro e raio:

```
public class Círculo {
 // variáveis de instância
    private float x, y, raio;

 // constante estática pi
    private static final float pi = 3.14159;

    // construtor
    public Círculo(float x, float y, float raio){
```



```
        this.x = x;
        this.y = y;
        this.raio = raio;
    }

    // métodos
    public float circunferência() {
        return (2 * pi * raio);
    }
    public float area() {
        return (pi * raio * raio);
    }
}
```

Neste caso, o uso de `this`, é obrigatório para evitar a ambigüidade entre as variáveis dos argumentos e as variáveis do objeto (seria dispensável se usássemos nomes distintos). Observe que não é declarado um *tipo de retorno* para o construtor. Ele implicitamente retorna uma instância da classe onde é definido.

Agora podemos criar vários círculos diferentes usando como gabarito a classe `Círculo` em uma única linha:

```
Círculo c1, c2, c3;

c1 = new Círculo(0.0f, 0.0f, 1.0f);
c2 = new Círculo(2.5f, 1.414f, 3.14159f);
c3 = new Círculo(2.0f, 2.0f, 2.0f);
```

Todos os objetos têm seus próprios métodos e variáveis. Uma operação sobre um dos objetos não afeta o outro:

```
float a1 = c1.area();  // a1 = 3.14159;
float a3 = c3.area();  // a2 = 12.56636
```

Múltiplos Construtores e Sobrecarga de Métodos

Podemos ter mais de um construtor para um objeto, assim como podemos ter formas diferentes de chamar um mesmo método. É possível *sobrecarregar* os nomes dos métodos e construtores para que, dependendo do *número* e *tipo* dos argumentos com que são chamados, eles realizem coisas (ou inicializações) diferentes. Isto é muito comum em linguagens orientadas a objetos e caracteriza um tipo de polimorfismo.

Quando criamos o novo construtor para `Círculo`, que necessita obrigatoriamente de três argumentos, *não podemos mais usar* aquele construtor

simples `Círculo()` a não ser que o redefinamos novamente. O sistema só dá construtores a quem não têm. Podemos fazer isto e ter dois construtores diferentes:

```
public class Círculo {
 // variáveis
    private float x, y, raio;

 // constante estática pi
    private static final float pi = 3.14159;

    // construtor 1
    public Círculo(float x, float y, float raio){
        this.x = x;
        this.y = y;
        this.raio = raio;
    }

    // construtor 2
    public Círculo(){
        this(0.0f, 0.0f, 1.0f);
    }

    // métodos
    public float circunferência() {
        return (2 * pi * raio);
    }

    public double area() {
        return (pi * raio * raio);
    }
}
```

`this(0.0f, 0.0f, 1.0f)` é uma chamada ao construtor *desta* classe que recebe três números de ponto flutuante como argumentos. É o mesmo que chamar `Círculo(0.0f, 0.0f, 1.0f)`. O segundo construtor então faz na verdade uma chamada ao primeiro construtor, com aqueles argumentos. Agora podemos voltar a criar círculos com o construtor simples:

```
Círculo c4 = new Círculo();  // Centro:(0,0); Raio 1
```

Construtor da superclasse

Antes falamos que o construtor chamava sempre o construtor da *superclasse*, mesmo que este não estivesse definido. O construtor chamado é sempre o construtor *default* (sem argumentos). Se estamos estendendo uma classe que tem

outros construtores, precisamos criar um construtor que realize esta chamada explicitamente. A chamada ao construtor da superclasse é feita através da palavra-chave super, por exemplo

```
public class CírculoOpaco extends Círculo {
 // variáveis
    private float x, y, raio;
    private Color cor;

    // construtor 1
    public CírculoOpaco(){
        super(0, 0, 1);
    }
}
```

O que o construtor *default* (aquele que ninguém vê) é na verdade o seguinte:

```
public SuaClasse(){
        super();
}
```

Se a classe que você estende não tem construtor *default*, você terá um erro de compilação que só será corrigido quando você definir explicitamente o construtor da sua classe.

Exercícios

5. Altere a classe `Círculo` fazendo com que suas variáveis `x`, `y` e `raio` sejam privativas à classe (use o modificador `private`). Agora a única forma de se definir um círculo é através de seu construtor. Crie três novos métodos então que permitam que se leia as coordenadas do centro e o raio.
6. Crie um método que aumente a dimensão do `raio` cada vez que é chamado. O tamanho do incremento deve ser passado como argumento do método.
7. Crie uma aplicação gráfica simples (pode ser baseada no `HelloWorld` Applet primeiro módulo) que desenhe um círculo na tela. O programa deve receber da linha de comando o raio do círculo, criar uma instância e usar os dados obtidos para desenhar o círculo na tela. Use o método `Graphics.drawLine` e a equação do círculo (no início deste capítulo). Você terá que definir coordenadas diferentes de 0,0 pois a origem da aplicação gráfica é seu ponto superior esquerdo.

Variáveis e Métodos de Classe

Toda variável em Java tem que estar contida dentro uma classe. Também não existem variáveis globais na linguagem. Como então armazenar dados que não sejam alteradas por cada instância de um objeto? Para contar objetos criados, por exemplo?

A solução é usar variáveis estáticas (ou variáveis de classe). Usando o modificador `static` na declaração de uma variável, esta será acessível através da própria classe, independente de objetos.

No exemplo abaixo, utilizamos uma variável de classe para contar o número de vezes que o construtor é acessado e poder estabelecer um número de série para cada Cessna criado. Este valor pode ser usado a qualquer momento para saber quantos aviões já foram criados. Uma variável desse tipo não poderia ser de instância pois seria inicializada a cada objeto criado.

```
public class Cessna {
 // variável de classe
    public static int numSérie = 0;

 // variável de instância
 public int serial;

 // Construtor
 Cessna() {
    (...)
    serial = numSérie++; // incrementa a cada
                         // chamada do construtor
 }

    (... restante da classe ...)
}
```

Podemos então, dentro de um programa de controle de estoque da fábrica de aviões por exemplo, saber quantos Cessnas já foram fabricados. A melhor forma de fazer isto é usando o nome da classe:

```
System.out.println("No de Cessnas criados: " +
       (Cessna.numSérie));
```

Você também pode usar os nomes dos objetos mas deve lembrar que se trata de uma variável estática. Um objeto pode alterar o valor que será lido por outro:


```
Cessna c1 = new Cessna();
Cessna c2 = new Cessna();
c1.numSérie = 100;
c2.numSérie = 120;
Cessna.numSérie = 200;
```

Todas as três últimas atribuições acima alteram a mesma variável. A última sintaxe é a mais indicada por ser menos confusa.

Um parêntese: veja que estrago pode ser causado por usar variáveis públicas! De repente o número de série deixou de servir para identificar aviões. Mantenha suas variáveis declaradas como `private`!

Os métodos que utilizamos em nossos exemplos até agora, com exceção do main(), foram métodos de instância, ou seja, atuavam sobre um objeto. Como fizemos com variáveis, podemos também definir métodos de classe, ou estáticos. Métodos estáticos se comportam como funções. Precisam de argumentos para operar (a não ser que operem somente localmente) pois não atuam sobre um objeto em particular. Os métodos de `java.lang.Math`, por exemplo, precisam ser invocados da forma:

```
Math.sqrt(2);
Math.abs(x + y);
```

Se definirmos um método tipo `static` em `Cessna`, para acessá-lo teríamos que fazer:

```
Cessna.métodoEstático();
```

Novamente, poderíamos usar uma instância, mas, além de ser algo desnecessário, torna a sintaxe confusa. Para usar métodos estáticos não precisamos criar objeto algum.

Exercício

8. Substitua o π usado no `Círculo` pelo valor mais preciso `Math.PI`, definido na biblioteca de classes (API) da linguagem Java. Para garantir a conversão de tipos (`double` para `float`, será necessário usar coerção (cast) explícita no retorno dos métodos. Ex:

   ```
   return (float)(2 * Math.PI * raio);
   ```

9. Crie um método:

   ```
   public float arco_x(float radianos) { ... }
   ```


que retorne a coordenada `x` do círculo correspondente ao arco do ângulo passado como argumento. Para calcular o seno do ângulo, use a função estática `Math.sin(float radianos)`. Use a equação: $x = raio * \text{sen}(\alpha)$;

10. Use a função `Math.random()` para gerar uma quantidade de 1 a 10 círculos de tamanhos aleatórios. Faça-os aparecer na sua aplicação gráfica. Registre a contagem dos círculos criados (use `Graphics.drawString()`).
11. Incremente o exercício anterior para que os círculos tenham posições diferentes também.

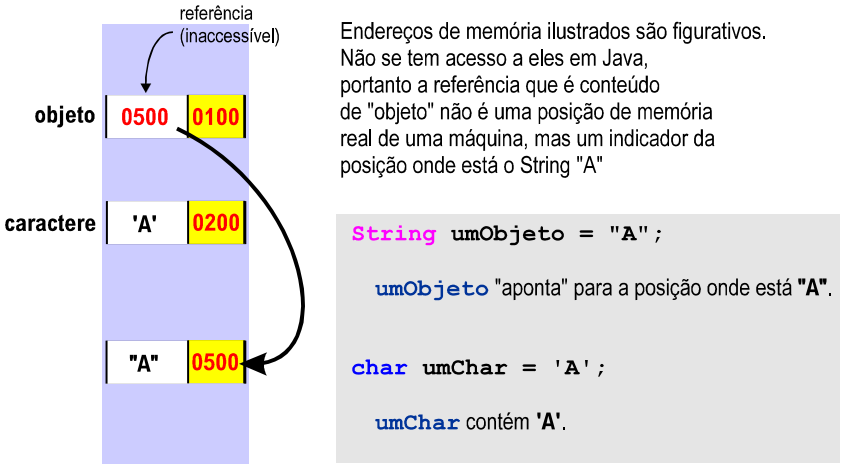
Referências e Apontadores

É amplamente divulgado que uma das principais características de Java que a difere de C com vantagens é a ausência de apontadores (ou ponteiros). Aí vem a pergunta: como, em uma linguagem sem apontadores, indexar vetores ou criar listas encadeadas?

Na verdade, Java tem sim apontadores, mas eles são internos à linguagem e não se pode manipulá-los da forma que se faz em C. Os apontadores de Java não têm um endereço que se possa ter acesso ou modificar. São implícitos e pode-se programar à vontade sem ter que se preocupar com a existência deles.

Os apontadores de Java são chamados de *referências*. Usamos várias delas neste capítulo. Sempre que se declara uma variável como sendo do tipo de uma classe (e não um tipo primitivo), cria-se uma referência a um objeto, por enquanto inexistente. Quando criamos o objeto (com new) e o atribuímos uma variável, ela passa a armazenar não o objeto em si, mas uma *referência* para ele.

Java então possui duas formas de armazenar informações em uma variável: diretamente ou indiretamente, através de referências. *Tipos primitivos* são passados diretamente, *por valor*. *Objetos* são *sempre passados por referência* e por isso são chamados de *tipos de referência*. A ilustração a seguir mostra este conceito.



Lembre-se que a cadeia de caracteres, ou *string*, é um objeto em Java. É construída a partir da classe `String`, que faz parte da biblioteca fundamental da linguagem. Já o tipo primitivo `char`, armazena caracteres individuais. A atribuição

```
String objeto = "A";
```

é na verdade um atalho para

```
String objeto = new String("A");
```

e só vale para a criação de objetos do tipo `String`. Outros objetos devem ser criados usando `new` e um construtor apropriado. Este atalho existe pelo fato da criação de cadeias de caracteres ser uma tarefa extremamente comum e pelo *status* especial dado aos literais do tipo `String`.

Como os tipos de referência não recebem o objeto em si, mas uma referência para ele, é possível haver mais de uma referência a um mesmo objeto e isto não for observado, poderá gerar erros. Veja o seguinte trecho de código:

```
Círculo cGrande, cPequeno; // declarações

// cria objeto
cPequeno = new Círculo(0.0, 0.0, 5.0);

// copia cPequeno em cGrande
cGrande = cPequeno;

// muda o raio de cPequeno
cPequeno.mudaRaio(2.0);

System.out.println("Raio de cGrande: " + cGrande.raio() +
                   "; Raio de cPequeno " + cPequeno.raio());
```


Se você incluir o trecho de código acima em um programa Java, irá esperar que ele imprima a linha a seguir, correto?

```
Raio de cGrande: 5.0; Raio de cPequeno: 2.0
```

Só que na realidade, ele vai imprimir outra coisa:

```
Raio de cGrande: 2.0; Raio de cPequeno: 2.0
```

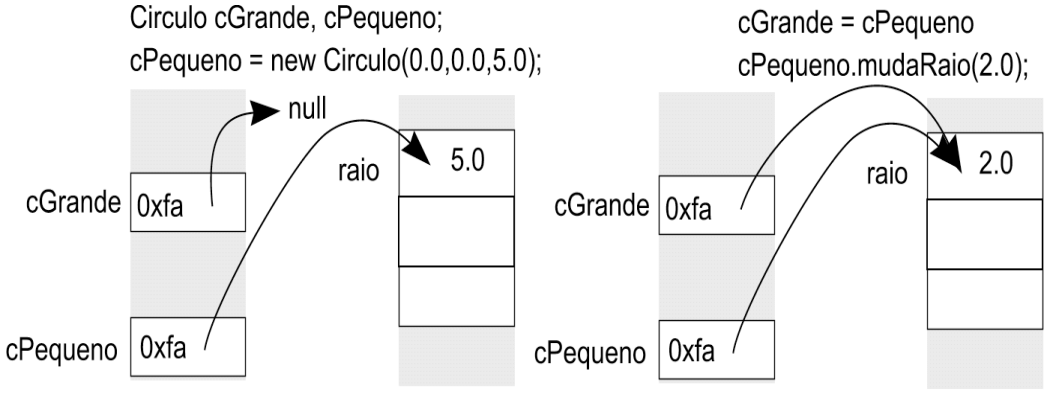
E agora? O que houve? Nós não havíamos guardado o valor anterior de `cPequeno` na variável `cGrande`?

A resposta é "Não!". Quando fizemos

```
cGrande = cPequeno;
```

nós guardamos sim o *conteúdo* de `cPequeno` em `cGrande` só que o conteúdo de `cPequeno` *não é* um círculo! O conteúdo de `cPequeno` é a *referência* para a posição onde está a um círculo de raio 5 e isto é o que foi copiado para a variável `cGrande`. Agora tanto `cPequeno`, como `cGrande`, *possuem a mesma referência* que aponta para o círculo de raio 2!

Alteramos o objeto apontado por `cPequeno`, mudando o seu raio. Como a referência de `cGrande` é a mesma de `cPequeno`, a mudança será vista em ambos ou seja, a informação *não foi* preservada. A figura a seguir ilustra todo o processo.



Vamos supor agora que um programa tem uma senha que será comparada com a senha digitado por um usuário. O trecho de código a seguir ilustra esta situação:

```
String senha, tentativa;
senha = new String("Carpe Diem");

    (...)
```



```
tentativa = "Carpe Diem";

        // tentativa armazenaria a senha
        // digitada pelo usuário.

if (tentativa == senha) {
   System.out.println("Seja Bem Vindo!");
   entre();

} else {
   System.out.println("Vá embora!");
}
```

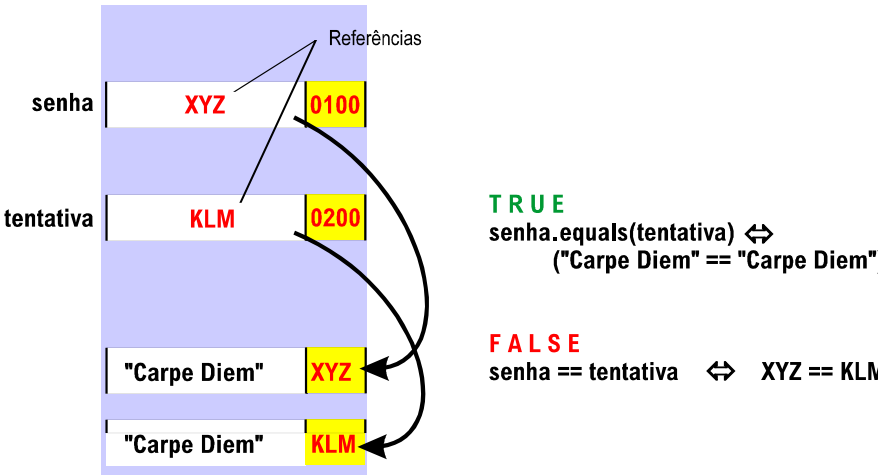
E então? Você roda o programa e...

```
Vá embora!
```

A situação é a mesma que foi descrita anteriormente, só que desta vez o erro aconteceu na comparação. A expressão:

```
tentativa == senha
```

é falsa porque as referências de `tentativa` e `senha` são diferentes, apesar de conterem *a mesma* cadeia de caracteres. As duas contém exatamente o mesmo texto, mas como estão em posições diferentes, e o que foi comparado aqui foram as referências, não o conteúdo dos objetos, a expressão resulta em `false`. A figura a seguir ilustra a situação.



A única forma de comparar objetos (ou fazer cópias deles) é através de métodos. A classe `String`, por exemplo, possui um método `equals()` somente para comparar duas cadeias de caracteres. Desta forma, poderíamos rescrever a expressão anterior da forma:

```
if (tentativa.equals(senha)) {
```

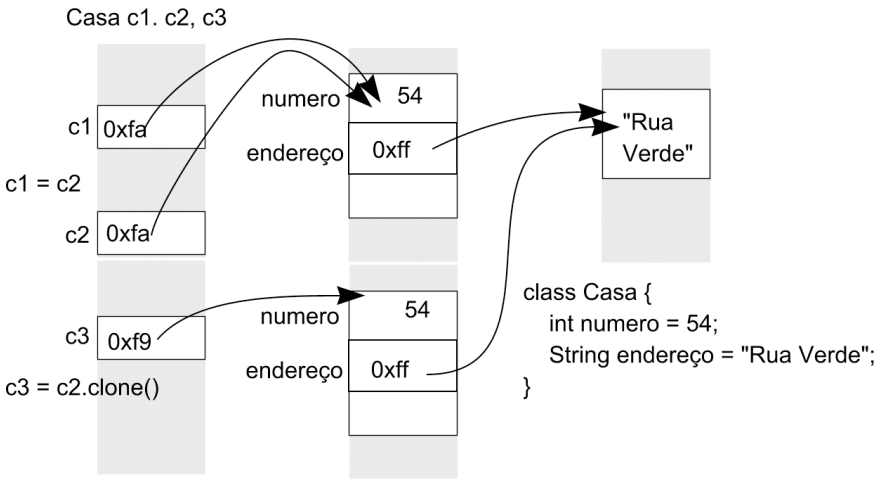

De forma semelhante, podemos usar o método `clone()` da classe `Círculo` para fazer cópias de objetos:

```
cGrande.clone(cPequeno);
```

resultando em duas variáveis com referências para objetos diferentes e com o mesmo conteúdo.

Os métodos `equals()` e `clone()` na verdade são definidos na classe `Object` e herdados por todas as classes. A versão de `String` é uma sobreposição do original para lidar com cadeias de caracteres. É importante sobrepor estes métodos em cada objeto que necessitar ser comparado ou duplicado.

O `clone` usado em círculo, por exemplo, só irá funcionar se `Círculo` redefinir o método `clone` (ou declarar `implements Cloneable`). Isto é porque `clone`, em `Object` é declarado `protected`, e como tal, só pode ser usado em superclasses. Ou Círculo redefine `clone` com modificador `public` ou implementa `Cloneable`, que é uma interface que já define `clone` como público. É sempre necessário redefinir `clone` quando um objeto possui campos de dados que são outros objetos. `clone`, como está definido em `Object`, apenas copia os valores dos campos do objeto. Se estes campos contiverem objetos, só as suas referências serão copiadas e não o seu conteúdo:



Como é que eu poderia comparar dois aviões Cessna? A implementação original (herdada de `Object`) provavelmente não servirá. O `equals()` para a classe `Cessna` deveria me dizer se um Cessna A e outro B correspondem ao mesmo avião. Isto só seria verdade se ambos tivessem o mesmo número de série. Poderíamos então implementar o método assim:


```
public boolean equals(Object outro) {
    if (this.serial == other.serial) return true;
    else return false;
}
```

Destruição de Objetos

Depois que vimos como criar objetos, resta agora saber como destruí-los e liberar espaço na memória, certo? Errado! Em Java, você não precisa se preocupar com a destruição de objetos! Uma das inovadoras características da linguagem é exatamente esta: a *coleta automática de lixo*, que libera automaticamente a memória que não é mais utilizada. Como é que o sistema de execução sabe que um objeto não vai ser mais utilizado? Simples: quando um objeto não têm mais referências apontando para ele, não pode ser mais usado então ele é descartado e a sua memória liberada. Por exemplo, suponha que eu crio um objeto qualquer:

```
StringBuffer sb;
sb = new StringBuffer("Odie");
```

A variável `sb` é uma referência para o objeto `StringBuffer` que contém o texto "Odie". Se agora fazemos:

```
sb = new StringBuffer("Garfield");
```

atribuímos a `sb` uma nova referência para outro objeto `StringBuffer`. O Odie ainda está lá, ocupando memória, mas logo que passar a "carrocinha" ele será recolhido, pois não tem dono.

O processo de coleta de lixo é realizado por uma linha de controle (*thread*) de baixa prioridade que fica monitorando o sistema. A maior parte do trabalho é realizado quando outras coisas não estão sendo feitas no programa, portanto, não há grande perda de desempenho.

Há um tipo de finalização porém que pode ainda ser necessária quando há arquivos ou *sockets*[3] abertos. Como esses recursos não são liberados pelo coletor de lixo, é necessário fazer a liberação "manualmente". Existe um método definido na classe `Object` (e portanto herdado por todas as outras classes) projetado para operações desse tipo: `finalize()`. Para usá-lo, é só sobrepor sua implementação original (vazia) com uma implementação específica. Ele é

[3] Conexões TCP/IP

chamado automaticamente pelo sistema de execução Java. Um exemplo para fechar um socket seria:

```
protected void finalize() throws IOException {
 if(socket != null) close();
}
```

Exercícios

12. Crie um método `equals()` para a classe `Círculo`. Qual o critério utilizado para comparar dois círculos?
13. O método `clone()` faz uma cópia de todas as referências que um objeto possui. Mas o `clone` herdado de `Object` pode não resolver todos os problemas. Se você tem um objeto que possui referências a outro objeto ou a um vetor, o clone não realizará uma cópia perfeita. Implemente uma versão de `clone()` pare resolver este problema.

1.3. Herança, Classes Abstratas e Interfaces

Todas as classes vistas nos exemplos apresentados até agora foram sempre extensões de uma classe já existente. Em alguns exemplos, isto esteve explícito através da cláusula "`extends`", seguida pelo nome da classe estendida, ou *superclasse*; em outros, a superclasse não fazia parte da declaração. Neste caso, ela *implicitamente* estendia a classe `Object` (`java.lang.Object`). Ou seja, dizer

```
class Cessna { ... }
```

é o mesmo que dizer

```
class Cessna extends Object { ... }
```

Para criar uma subclasse de qualquer classe já existente e acessível, basta declará-la como tal:

```
class Estrela extends Astro
```

A subclasse herda todas variáveis e métodos públicos e protegidos definidos na superclasse. Ela não herda os construtores mas ganha um construtor *default* e pode definir novos construtores e invocando os construtores da superclasse.

A extensão de classes caracteriza a *herança* e torna possível o *polimorfismo*, caracterizado principalmente pela *sobreposição de métodos*, analisada neste capítulo. Um objeto polimórfico pode assumir múltiplas formas, seja com as

características de sua própria classe, seja com características da classe que estende. A extensão de uma classe geralmente adiciona uma nova funcionalidade à classe estendida. A nova classe herda métodos e campos de dados públicos e protegidos, adiciona novos métodos e variáveis e possivelmente redefine parte das funções da superclasse.

Na última seção, nos concentramos na característica da *abstração*, definindo classes como tipos de dados abstratos, e no *encapsulamento*, protegendo os dados e publicando uma interface pública de funções para classes e objetos. Nesta seção, iremos estudar como Java implementa não só a herança e polimorfismo, mas também a *modularidade*, realizada através dos pacotes.

As classes abstratas e interfaces são peças fundamentais no desenvolvimento de sistemas com alto grau de reutilização de código. As mais complexas APIs da linguagem Java dependem de interfaces e classes abstratas para alcançarem um alto grau de versatilidade e facilidade de uso. Veremos neste capítulo como usar interfaces e classes abstratas em aplicações Java.

Pacotes

Além da hierarquia formada pelas classes e superclasses, Java oferece mais uma outra forma para organizar conjuntos de classes: os *pacotes*.

Você pode agrupar qualquer quantidade de classes relacionadas ou não em um pacote. As classes não precisam ter relação alguma entre si, embora isto seja comum quando se define qualquer hierarquia.

Todos os programas que fizemos até agora estavam dentro de pacotes! Não declaramos nada, então Java colocou nossas classes em um pacote *default*, restrito ao diretório onde compilamos e rodamos nossos arquivos. E pacotes são mais ou menos isto: *diretórios*. Quando definimos um pacote para agrupar classes em Java estas classes são armazenadas em diretórios específicos.

No *Java Development Kit (JDK)*, pacotes são utilizados para agrupar as classes da biblioteca fundamental da linguagem Java (API), por exemplo:

```
java.awt
java.lang
java.io
java.awt.event
```


são todos nomes de pacotes. Dentro deles estão as classes. Freqüentemente temos que usar uma ou outra classe de um desse pacotes e então importamos o mesmo, usando

```
import java.awt.*;
```

por exemplo, para ter acesso à classe `Frame`, `Button`, `Graphics`, etc.

Uma aplicação ou módulo de aplicação provavelmente usaria pacotes para organizar todas as suas classes. É um recurso a mais que o programador tem à disposição para realizar a organização de informação, e assim simplificar o seu uso.

Como criar um pacote

Para formar um pacote, basta declará-lo no início de cada classe que o compõe. O conjunto formado pelas *classes que declaram pertencer a um mesmo pacote* caracterizam o mesmo. Para isto é necessário que cada programa tenha como primeira instrução a declaração

```
package nome-do-pacote;
```

O nome de um pacote pode ser qualquer identificador válido. Na ausência dessa declaração, a classe é incluída em um pacote *default*, genérico, que na prática contém todas as classes que não tem package definido e que estão no mesmo subdiretório de trabalho.

Se estivéssemos escrevendo um simulador de vôo, poderíamos colocar a classe `Cessna` em um pacote `simulador`, para organizá-la junto com outras classes relacionadas com o programa de simulação de vôo, como `Aeroporto`, `Nuvens`, etc. e ter privilégios de *acesso especial* à métodos e variáveis de outras classes pertencentes ao mesmo pacote. No arquivo, colocaríamos:

```
package simulador;

public class Cessna { ... }
```

Agora a classe se chama `simulador.Cessna`.

Como usar um pacote

Para usar classes pertencentes a um pacote declarado, é necessário usar a combinação

```
nome_de_pacote.Nome_de_Classe
```


sempre que se precisar usar a classe, porque agora o nome da classe é a combinação do nome do pacote com o nome original declarado. Uma forma mais prática de usar uma classe, principalmente quando uma mesma classe vai ser usada várias vezes é importá-la através da declaração `import`:

```
import nome_de_pacote.Nome_de_Classe;  ou
import nome_de_pacote.*;
```

que permite que se use o nome *abreviado* da classe em vez do *nome longo*. `import` não é uma declaração necessária e pode ser dispensada sempre. Pode-se usar qualquer classe sem precisar de `import`, chamando-a pelo nome completo:

```
class AplicacaoGrafica extends java.awt.Frame {
    public static void main(java.lang.String args[]) {
        java.awt.Frame fr = new java.awt Frame("Frame");
        java.awt.Button b = new java.awt.Button("Botão");
        b.addActionListener(new java.awt.event.ActionListener() { ...
```

Usando dois *imports*, o codigo fica muito mais legível:

```
import java.awt.*;
import java.awt.event.ActionListener;

class AplicacaoGrafica extends Frame {
   public static void main(String args[]) {
        Frame fr = new Frame("Frame");
        Button b = new Button("Botão");
        b.addActionListener(new ActionListener() { ...
```

A classe `String` não precisa de `import` porque é parte do pacote `java.lang`, que o sistema já importa automaticamente.

O grande benefício aqui é que esta nova nomenclatura evita conflitos entre nomes de classes. Os fabricantes de APIs Java sempre distribuem todas suas classes dentro de pacotes e, se você encontra dois pacotes com uma classe de mesmo nome ainda assim pode usá-las no mesmo programa desde que use o nome totalmente qualificado para uma delas.

```
import com.microsoft.win32.Button;

(...)

Button bm = new Button("Botão Microsoft");
java.awt.Button bj = new java.awt.Button("Botão Java");
```


Pacotes, diretórios e CLASSPATH

A declaração `import` na verdade não importa uma classe, mas *importa* o caminho para que o interpretador a encontre e possa saber de quem se trata quando o programador não usar o nome completo da classe.

A declaração `package` é apenas parte da construção de um pacote. Depois da compilação, as classes que constam do pacote devem ser colocadas em um diretório que tenha o mesmo nome que o pacote. Quando o interpretador Java precisar importar uma classe pertencente a um pacote, ele irá procurá-la dentro desse diretório. O caminho até este diretório deve também ser conhecido pelo interpretador. Para isto, deve estar na lista de diretórios que ele usa para procurar classes, que é o `CLASSPATH`.

A `CLASSPATH` é uma variável de ambiente do sistema definida pela máquina virtual Java (JVM). Na hora em que você instala uma aplicação Java, ela poderá definir uma `CLASSPATH` antes que possa rodar.

O interpretador só procura classes dentro de pacotes que estejam acessíveis pelo `CLASSPATH`. Por exemplo, se você tem várias classes declaradas como parte de um pacote chamado `livro` e uma variável de ambiente (no *Windows*) definida no `AUTOEXEC.BAT` como :

```
CLASSPATH=C:\programas\
```

o interpretador procurará suas classes em:

```
.\livros\
C:\jdk1.2\jre\lib\rt.jar\livros\
C:\programas\livros\
```

O sistema já define previamente um `CLASSPATH` básico que inclui as classes fundamentais Java (`rt.jar`) e o diretório atual.

Dentro de pacotes pode haver classes e outros pacotes. Pode-se ter grandes hierarquias de pacotes. Na API Java, por exemplo, todos os pacotes `java.lang`, `java.awt`, `java.io`, etc estão dentro de um pacote chamado java. Há um pacote chamado `java.awt.event` dentro de `java.awt`. Se você abrir e visualizar o conteúdo do arquivos `rt.jar` (que contém as classes *run-time* Java no JDK), verá que cada subpacote representa um diretório. O pacote `java.awt.image` representa

```
%CLASSPATH%\java\awt\image
```


Se uma classe declara que está em um certo pacote, ela *deve estar* fisicamente dentro do diretório e subdiretórios correspondentes para permitir que o interpretador Java a encontre. Os pontos entre os pacotes são traduzidos em separadores de caminho de diretórios. Se você declara:

```
package BR.com.empresa.ferramentas.gui;
```

antes de uma definição de classe, você deve colocar suas classes em um subdiretório

```
%CLASSPATH%\BR\com\empresa\ferramentas\gui\
```

A convenção de usar o nome de domínio ao contrário é recomendado pela especificação da linguagem Java como um meio de garantir que nunca haja conflito entre nomes de classes entre fabricantes diferentes.

Se você usar o compilador `javac` com a opção `-d`, ele criará os diretórios necessários e moverá os arquivos compilados automaticamente para seus lugares:

```
javac –d C:\PROGRAMAS Cessna.java
```

cria automaticamente um diretório `simulador`, abaixo de `C:\PROGRAMAS` e copia as classes resultantes da compilação para lá. Você terá que incluir, depois, `C:\PROGRAMAS` no `CLASSPATH` para que o interpretador encontre os arquivos quando precisar. Você também pode fazer o mesmo usando o diretório atual:

```
javac –d . Cessna.java
```

que criará `simulador` abaixo do diretório atual.

OBSERVAÇÃO IMPORTANTE: *Apesar da solução acima ser prática, ela pode trazer alguns problemas na hora em que você for compilar outras classes que usam programas de outro pacote. Depois que você compilar seus programas-fonte para um outro pacote,* remova-os *do seu diretório de trabalho. O compilador irá reclamar se ele achar os arquivos .java do pacote que você está usando no diretório atual. Para evitar esses problemas,* não use *a opção* –d *e crie seus diretórios manualmente, compilando os programas-fonte dentro deles. Isto não é problema se você usa um ambiente integrado de desenvolvimento diferente do JDK, como o JBuilder ou Visual Café Pro.*

Não é possível importar pacotes; apenas se importa classes. Portanto fazer:

```
import java.awt.*;
```

no início de um programa importa *todas as classes* do pacote `java.awt` como `java.awt.Frame`, `java.awt.Graphics`, mas *não* importa as classes que estão dentro de `java.awt.event`. Para importar essas classes também, é preciso fazer:


```
import java.awt.*;
import java.awt.event.*;
```

A declaração `package` tem que ser a primeira coisa que aparece em um arquivo .java, se estiver presente. Só pode haver comentários antes dela. Depois de `package` podem vir nenhuma, uma ou mais declarações `import`, e, finalmente, uma ou mais definições de classes ou interfaces. Se essas três construções de primeiro nível estiverem presentes em um arquivo .java, elas devem seguir precisamente esta ordem:

- declaração de pacote
- declaração(ões) import
- definição(ões) de classe(s) e/ou interface(s)

Modificadores de Acesso

Os modificadores de acesso definem se uma variável ou método pode ser acessada por outras classes, por subclasses, por nenhuma outra classe ou por classes do mesmo pacote. Valem somente para variáveis de classe ou de instância. Não podem ser usados por variáveis locais.

Os modificadores básicos são `public` e `private`, que vimos nos capítulos anteriores. Quando nenhum modificador de acesso aparece na declaração de um método ou variável de instância ou de classe, o acesso é permitido apenas *dentro do mesmo pacote*. Este modificador oculto é definido na especificação Java como "friendly" ou "package". Há ainda um modificador chamado `protected` que é usado para ampliar o acesso de métodos e variáveis às subclasses.

Um bom projeto de classe sugere que as variáveis e métodos tenham somente o nível de acesso necessário ao uso a que se destina. Isto torna a classe fácil de usar e garante maior segurança. Se uma classe só vai usar uma variável para operações locais, esta deve ser declarada como `private`. Se um método só deve ser usado pela própria classe, classes do mesmo pacote ou subclasses, deve ser declarado como `protected`.

Com esses modificadores, pode-se controlar o aspecto das interfaces de um objeto ou classe. A interface pública é o conjunto de métodos disponíveis a todos. A interface protegida é a funcionalidade proporcionada às subclasses. A interface *amigável*, expõe funções a classes "amigas", que compartilha o mesmo pacote.

Os modificadores de acesso disponíveis na linguagem Java estão listados abaixo com o nível de acesso em ordem decrescente:

- **`public`**: Torna o método ou variável universalmente acessível.

  ```
  public int x;
  ```

  Um bom projeto orientado a objetos raramente usa variáveis públicas (o acesso às variáveis sempre pode ser feito através de métodos). Use para métodos, variáveis globais ou constantes que devem ser visíveis em todo lugar.

- **`protected`:** O acesso é permitido somente às subclasses e classes do mesmo pacote.

  ```
  protected int y;
  ```

  Classes que não fazem parte de nenhuma das duas famílias não tem acesso. Use no desenvolvimento de classes incompletas, projetadas para serem estendidas; em métodos e variáveis que devem ser acessíveis somente nas subclasses e pacotes e não fora deles. Observe que se você não declara pacote, sua classe pertence ao pacote *default* e `protected` age como `public` para as classes do mesmo diretório.

- *package ou friendly* (*default*): Quando *não há* modificadores de acesso explícito, subentende-se que o acesso é restrito às classes do mesmo pacote.

  ```
  int z; // não há modificador explícito!
  ```

  Neste caso, apenas as classes que tem a mesma declaração `package` podem ter acesso. A proteção é maior que `protected`. Se uma subclasse não pertencer ao pacote, ela não tem acesso, mesmo que seja subclasse. Assim como ocorre com `protected`, caso a classe *não tenha* declaração `package`, o acesso será permitido *à todas* as classes do mesmo diretório que também não têm esta declaração. Use quando métodos ou variáveis só interessarem à própria aplicação ou módulo (pacote) dela.

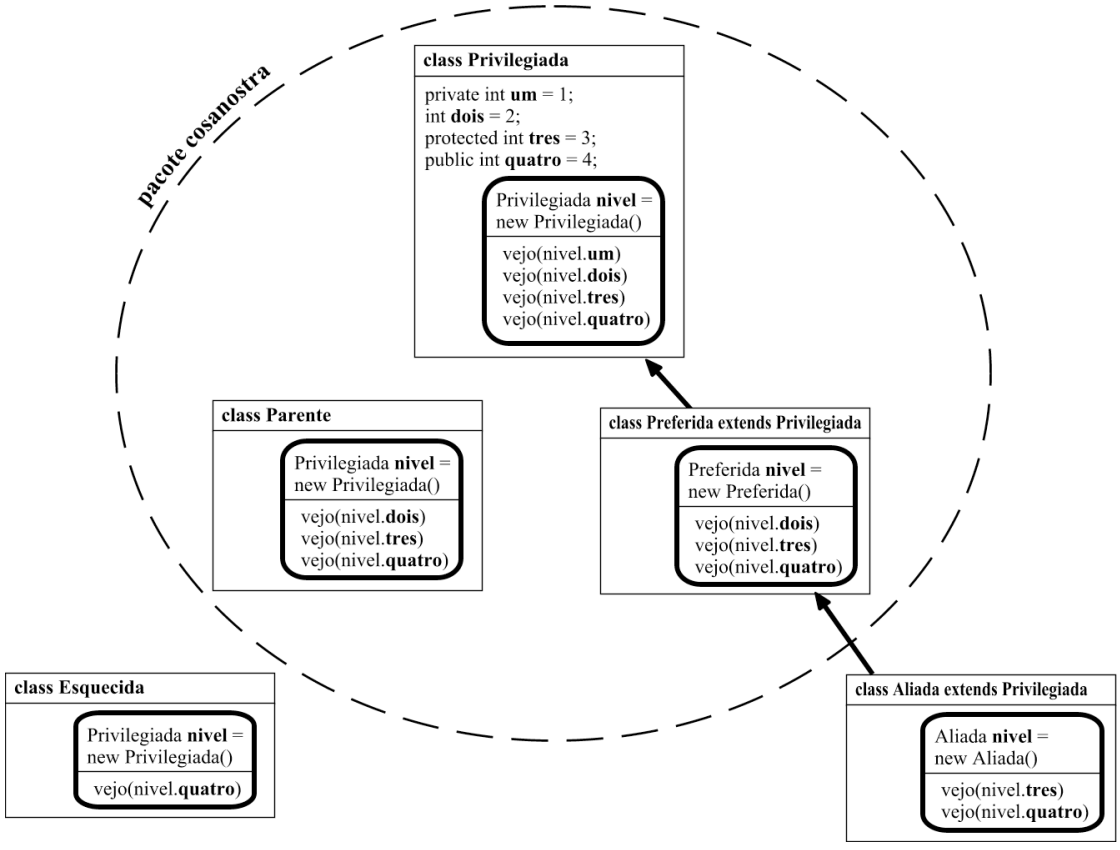
- **`private`:** Torna o método ou variável privativo à classe (se for `static`) ou objeto.

  ```
  private int k;
  ```


Os métodos e variáveis declarados com este modificador só podem ser usados internamente. Use para métodos e variáveis que só são usados dentro da classe, e não interessam à classes externas.

Agora que vimos as diferenças entre modificadores de acesso em *mil palavras*, vejamos tudo resumido na figura abaixo. As linhas pontilhadas separam pacotes diferentes. Os retângulos representam classes. Os retângulos de linha grossa e cantos arredondados representam objetos. As setas apontam para as superclasses.

Observe que o uso dos campos `protected` é feito pela própria subclasse, assumindo a sua posição de ser herdeira das variáveis das superclasses. As classes que não herdam os valores (`Esquecida` e `Parente`) são obrigadas a criar uma instância de uma classe que os possui para poder ter acesso.



Sobreposição de métodos ocultação de dados

Podemos ampliar ou alterar a funcionalidade de uma classe estendida através da *sobreposição* de seus métodos ou através da *sobrecarga* dos nomes dos métodos. A

sobrecarga já vimos no capítulo anterior com construtores. Funciona da mesma forma com métodos comuns. A diferença é que no caso dos métodos, a sobrecarga pode ocorrer por herança: se uma subclasse herda um método e define outro com mesmo nome e assinatura diferente, ela passa a ter dois métodos com o mesmo nome, o que caracteriza uma sobrecarga.

A sobreposição tem um efeito mais amplo sobre a funcionalidade do objeto. Consiste em *substituir* a implementação do método da superclasse com uma nova definição. As assinaturas, neste caso, devem ser *idênticas* ou serão interpretadas como um novo caso de sobrecarga.

Um método novo pode ter diferentes modificadores de acesso desde que estes garantam *mais* acesso que o método sobreposto. Um método declarado `protected` na superclasse pode ser declarado `protected` ou `public`, mas não `private` ou *friendly* (o modificador *default*) pois os últimos são mais restritivos. Os métodos também não podem declarar que provocam mais exceções que os métodos sobrepostos.

A sobreposição dos dados em variáveis e referências não os substitui. Eles continuam a existir mas a nova definição os torna ocultos. Na prática, isto pode ser visto como uma sobreposição. Eles ainda podem ser recuperados usando a palavra-chave super:

```
class Vinte {
   protected int num = 20;
}

class Duzentos extends Vinte {
   protected int num = 200;

   public static void main(String a[]) {
      System.out.println("Total: " + num + super.num);
   }
}
```

Invocação de métodos e ligação dinâmica

Quando um método é chamado sobre um objeto, o tipo daquele objeto em tempo de execução é quem diz qual implementação será usada. Você pode declarar um objeto como sendo do tipo de uma superclasse, por exemplo:

```
Object obj = new Cessna();
```

Isto é perfeitamente válido. Na hora em que o programa fizer

```
obj.toString();
```


não será o `toString()` definido em `Object` que irá ser chamado, mas o que está na memória naquele momento (objetos são alocados dinamicamente em tempo de execução), portanto, o `toString()` irá invocar o método em um objeto do tipo `Cessna`, e fazer o que foi definido na classe `Cessna`.

A única exceção a esta regra é quando em vez de se usar a referência a um objeto, se usa a palavra-chave `super`. A invocação `super.método()` *sempre* usa a implementação de um método de acordo com a superclasse e não de acordo com uma implementação sobreposta. O exemplo ("The Java Programming Language: p.61" - Gosling, Arnold - 1996) a seguir ilustra esta situação:

```
class That {
   /** retorna o nome da classe */
   protected String nm() {
      return "That";
   }
}

class More extends That {
   protected String nm() {
      return "More";
   }
   protected void printNM() {
      That sref = super;
      System.out.println("this.nm() = " + this.nm());
      System.out.println("sref.nm() = " + sref.nm());
      System.out.println("super.nm() = " + super.nm());
   }
}
```

A saída do programa acima é:

```
this.nm() = More
sref.nm() = More
super.nm() = That
```

Quando não estender uma classe?

Quando estender uma classe? Será que porque uma `Zebra` parece uma `Mula` poderíamos estender `Mula`, acrescentar umas listras e obter uma `Zebra`?

O processo de escolha de classes e objetos, a determinação das relações entre eles é um tópico complexo que têm resultado em milhares de publicações. A análise e projeto de sistemas orientados a objetos é um assunto importantíssimo. A sua aplicação no desenvolvimento de programas em Java ou em qualquer outra linguagem orientada a objetos é a melhor maneira de garantir

que um sistema vai atender às especificações da forma mais eficiente possível. No processo de desenvolvimento de um software, a implementação é a última etapa. A maior parte do esforço é despendido na análise do problema e posterior elaboração do modelo de objetos. Somente depois que todas as relações entre objetos são mapeadas é que tem início a implementação.

Não faz parte do escopo deste curso ensinar análise e *design* orientado a objetos, mas se você quiser levar à sério o desenvolvimento usando Java, não deve deixar de considerar a possibilidade de conhecer mais sobre esse assunto, essencial em cursos mais avançados sobre a linguagem Java e no desenvolvimento de software.

Para ilustrar um pouco a importância da análise cuidadosa do problema antes da sua implementação, vejamos um exemplo de implementação problemática.

Suponha que precisamos de uma classe que represente uma esfera. Poderíamos criar uma classe `Esfera` do zero. Mas, lembrando que um círculo parece-se com uma esfera e concluindo que muito do que já foi definido para `Círculo` poderia ser aproveitado em `Esfera` (duas coordenadas, alguns métodos, etc.), decidimos utilizar as propriedades das linguagens orientadas a objetos e fazer com que `Esfera` seja uma especialização de `Círculo`.

Fazemos então:

```
public Class Esfera extends Círculo { ... }
```

Como os métodos de `Círculo` são públicos, eles são todos herdados por `Esfera`. As variáveis não são vistas da nova classe porque são privativas. O ideal é que elas fossem `protected`, para que pudessem ser vistas pelas suas sub-classes. Vamos supor então, só para este exemplo, que as variáveis da classe Círculo são `protected`. Agora já podemos calcular a circunferência de um objeto `Esfera esf`:

```
float circ = esf.circunferência();
```

`Esfera` poderá definir seus próprios métodos e variáveis. Por exemplo, pode-se criar um método `volume()`:

```
public float volume() {
        return (3.14159 * raio * raio * raio);
}
```


Pode-se também sobrepor métodos da superclasse com uma nova implementação:

```
public double area() {
 return 4 * 3.14159 * r*r /3;
}
```

Também devemos ter um novo construtor para construir esferas, já que usamos uma dimensão a mais:

```
public Esfera(float x, float y, float z, float raio) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.raio = raio;
}
```

No início dissemos que o exemplo acima era problemático, mas aparentemente ele funciona sem problemas. Qual é o problema em utilizá-lo, então? O problema surgirá quando esta classe (`Esfera`) for utilizada por outras classes e métodos que trabalham com `Círculos`. O compromisso assumido quando se cria uma classe é que a sua interface pública (tudo o que está acessível para o público) vai sempre funcionar da forma esperada e assim podemos usar o objeto sem problemas sem se preocupar com as subclasses novas que possam surgir. As subclasses devem ser sempre especializações, isto é, diferentes formas assumidas pela superclasse. Não deve haver problemas em receber objetos `Fusca`, `BMW` e `Vectra` em um objeto `Estacionamento` criado para acomodar objetos do tipo `Automovel`. Eles são automóveis e, portanto, devem estender `Automóvel`. O problema é que uma esfera *não é* um círculo, portanto não deveria ser definida como subclasse de um.

Que problemas isto poderia gerar? Suponha que temos um objeto que construa rodas a partir de círculos. Poderia ter um construtor, por exemplo:

```
class Roda { ...{
    Roda (Círculo c1) { ... }
...}
```

Qualquer subclasse de `Círculo` é considerada um círculo e portanto o construtor de `Roda` poderia receber como argumento: `CírculoAzul`, `CírculoComBorda`, etc. Como a implementação da classe `Esfera`, no nosso exemplo, diz que ela "é" um círculo, ela poderia ser usada também:


```
Esfera e1;
Roda rodaBicicleta = new Roda(e1);
```

E então, quando o programa for calcular a área, por exemplo, que deveria ser igual em qualquer círculo, o programa irá falhar, pois a área de `Esfera` não é área de círculo.

É possível evitar este tipo de erro no estágio de projeto do modelo orientado a objetos, observando se há uma relação *é um* entre classes e subclasses. Neste estágio, é possível descobrir que `Esfera` não *é um* `Círculo` e evitar complicações que irão aparecer em estágios posteriores do desenvolvimento. A sobreposição de métodos para fazer coisas totalmente diferentes também é uma dica para detectar erros de projeto.

Podemos evitar que as subclasses alterem métodos fundamentais de uma classe usando o modificador `final`. Se tivéssemos projetado a classe `Círculo` com a seguinte declaração no método área:

```
public final double area() { ... }
```

Não seria mais possível que a classe `Esfera` sobrepusesse este método.

Uma Aplicação Orientada a Objetos

Nada melhor que um exemplo, para fixar os conceitos apresentados. Vamos mostrar na prática o uso de classes, superclasses e objetos, aplicando o conceito de polimorfismo (um diretório *é um* arquivo) e de composição (uma lista de arquivos *é parte de* um diretório). As classes do nosso exemplo fazem parte do pacote `sa` e são as seguintes:

- **Arquivo**: uma classe que representa um arquivo qualquer em um sistema hipotético.
- **Diretório**: um tipo especial de arquivo capaz de conter listas de arquivos.

As listagens, com comentários, seguem. Logo após há um breve explicação das principais partes de cada uma. Inicialmente, a classe `Arquivo`:

```
package sa;

/* Esta classe representa um arquivo
 * em um sistema de arquivos hipotético, semelhante ao
 * Windows 95 ou Unix */
public class Arquivo implements Cloneable {
```



```
///// Variáveis de classe /////
private static int inode; // guarda o número de arquivos criados
private static int cluster = 512; // tamanho minimo de arquivo

///// Variáveis de instância /////
private String nome; // nome do arquivo
private String extensão;  // extensão do arquivo
private int tamanho; // tamanho (bytes) do arquivo
private int id;      // identificação do arquivo
private Diretorio pai; // diretório pai

///// Construtores /////
/** Construtor completo */
public Arquivo(String nome,   String extensão,
               Diretorio pai, int tamanho) {
   this.nome = nome;
   this.pai = pai;
   if (pai != null) {
      pai.adiciona(this);
   }
   this.extensão = extensão;
   this.tamanho = ((int)(tamanho/cluster) + 1) * cluster;
   id = inode++;
}

/** Construtor para arquivos sem extensão */
public Arquivo(String nome, Diretorio pai, int tamanho) {
   this(nome, "", pai, tamanho);
}

/** Construtor para arquivos vazios somente para subclasses
    e classes deste pacote */
protected Arquivo(String nome, Diretorio pai) {
   this(nome, "", pai, cluster);
}

///// Métodos de instância /////
/**
 * Retorna o nome do arquivo
 */
 public String getNome() {
   return nome;
 }
 /** Retorna o pai
 */
 public Diretorio getPai() {
   return pai;
 }

/**
 * Retorna o tamanho do arquivo
 */
 public int getTam() {
```



```
    return tamanho;
  }

 /**
  * Retorna a extensão do arquivo
  */
  public String getExt() {
    return extensão;
  }

  /**
  * Muda o nome do arquivo
  */
  public void setNome(String novoNome) {
    nome = novoNome;
  }

 /**
  * Muda a extensão do arquivo
  */
  public void setExt(String novaExt) {
    extensão = novaExt;
  }

 /** Muda tamanho do arquivo (somente para classes
  *  amigas ou subclasses
  */
  protected void setTam(int novoTam) {
    tamanho = novoTam;
  }

  /**
  * Retorna caminho de subdiretórios
  */
  public String caminho() {
     String ext = "." + extensão;
     if (this instanceof Diretorio) {
          ext = "/";
     }
     if (pai != null) {
       return pai.caminho() + nome + ext;
     } else return "/";

  }

  ///// Métodos sobrepostos /////
 /**
  * Sobrepõe Object.toString()
  */
  public String toString() {
    String ext = "." + extensão;
    if (this instanceof Diretorio) {
       ext = "/ ";
```



```
        }
        return "[" + id + "] " + nome + ext + "\t(" + tamanho + ")";
      }

     /**
      * Sobrepõe Object.equals()
      */
      public boolean equals (Object obj) {
        if ( obj instanceof Arquivo ) {
           Arquivo arq = (Arquivo) obj;

           if ( arq.id == this.id ) {
              return true;
           }
        }
        return false;
      }
     /**
      * Sobrepõe Object.clone()
      */
      public Object clone() {
        try {
           return super.clone(); // retorna cópia do arquivo
        } catch (CloneNotSupportedException e) {
           throw new InternalError(e + "");
        }
      }
}
```

A classe `Arquivo` define o estado e comportamento de um arquivo. Um arquivo é um objeto que pode ser reproduzido (copiado), por isto ele implementa `Cloneable`. Um arquivo tem uma identidade, um nome, uma extensão, um diretório pai e um tamanho. Um arquivo possui métodos que permitem ler e alterar algumas de suas variáveis, imprimir suas características e compará-lo com outro arquivo.

A forma de comparar um arquivo com outro é diferente da forma de comparar um objeto genérico, portanto o método `equals` foi redefinido. A forma de imprimir as informações de um arquivo também é diferente da forma de imprimir as informações de um objeto genérico, então `toString` foi redefinido. Isto permite que o arquivo seja usado em qualquer lugar onde um `Object` for aceito e poderá invocar os métodos `toString` e `equals`. O método `clone` é sobreposto para implementar a interface `Cloneable` e garantir que o arquivo pode ser copiado.

No construtor é verificado se o diretório pai é `null`. Todo arquivo têm um diretório pai a não ser que este arquivo seja a própria raiz do sistema. Se o arquivo em questão não for a própria raiz, ele pode ser adicionado a um diretório pai. Para isto, é invocado um método de `Diretorio` que adiciona o próprio objeto atual (`this`), que é um arquivo, ao seu diretório pai.

O método `toString` verifica se o objeto atual é uma instância de `Diretorio`, usando o operador `instanceof`. Como todo diretorio é um arquivo, ele também pode invocar `toString` e a verificação garante que o que vai ser impresso é diferente se o objeto for um `Diretorio`.

A listagem a seguir contém a classe `Diretório`:

```
package sa;

import java.util.Vector;  // array dinamico

/* Esta classe representa um diretório
 * em um sistema de arquivos*/

public class Diretorio extends Arquivo {

   private Vector lista;

   ///// Construtores /////

   /** Constroi um diretório */
   public Diretorio(String nome, Diretorio pai) {

      super(nome, pai); // esta linha é necessária!
      lista = new Vector(2);
   }

   ///// Métodos de instância /////

   /**
    * Adiciona novo arquivo na lista
    */
    public void adiciona(Arquivo arquivo) {
      lista.addElement(arquivo);
      setTam(this.getTam() + arquivo.getTam());
    }

   /**
    * Tira arquivo da lista
    */
    public void remove(Arquivo arquivo) {
      lista.removeElement(arquivo);
      setTam(this.getTam() - arquivo.getTam());
```



```
 }

/**
 * Retorna vetor com conteudo do diretorio
 */
 public Arquivo[] dir() {
   Arquivo[] dirList = new Arquivo[lista.size()];
   lista.copyInto(dirList);  // copia Vector em vetor dirList
   return (Arquivo[])dirList;
 }

 /**
 * Lista conteúdo com filtro de extensão
 */
 public Arquivo[] dir(String ext) {
   Vector listaFilter = new Vector(lista.size());
   for (int i = 0; i < lista.size(); i++) {
      Arquivo arq = (Arquivo)lista.elementAt(i);
      if (arq.getExt().equals(ext)) {
         listaFilter.addElement(arq);
      }
   }
   Arquivo[] dirList = new Arquivo[listaFilter.size()];
   listaFilter.copyInto(dirList);  // copia Vector em dirList
   return (Arquivo[])dirList;
 }


 ///// Métodos sobrepostos /////

 /**
 * Sobrepõe Arquivo.toString()
 */
 public String toString() {
   return super.toString() + " " + lista.size() + " arquivos";
 }



 /**
  * Sobrepõe Arquivo.clone()
  * É necessário pois a cópia de um diretório não copia
  * a lista de arquivos que ele contém (passa a ser
  * simplesmente um atalho para o mesmo lugar)
  */
 public Object clone() {
   Diretorio d = (Diretorio)super.clone(); // copia diretorio
   d.lista = new Vector(lista.size()); // cria novo Vector vazio

   // copia arquivos um por um para nova localidade
   for (int idx = 0; idx < this.lista.size(); idx++) {
      Arquivo arq = (Arquivo)lista.elementAt(idx); // le da lista
      Arquivo dolly = (Arquivo)arq.clone(); // copia arquivo
```

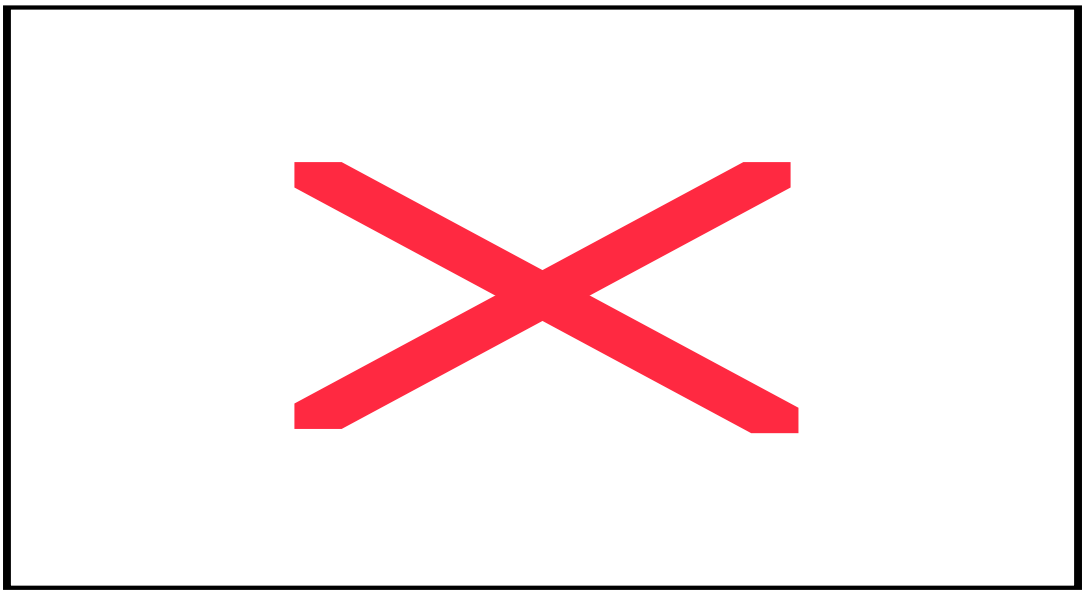


```
            d.lista.addElement(dolly); // adiciona no Vector
          }
          return d;  // devolve novo diretório
        }
    }
```

Um `Diretório` é um `Arquivo`. Ele tem uma lista de arquivos, que aqui é representada através de um objeto do tipo `Vector`. Vector é um vetor de dimensões que podem variar dinamicamente. `Vector` armazena a lista de arquivos contida no diretório. Um diretório possui métodos para adicionar, remover e listar os arquivos que contém. Uma sobrecarga do método `dir` (lista), permite ainda que a listagem seja feita filtrando a extensão do arquivo.

`Diretorio` sobrepõe `toString` porque um diretório possui informações diferentes para exibir como a quantidade de arquivos que possui. Mas, o resto das informações é semelhante, então ele invoca `super.toString` para concluir o trabalho.

`Diretório` sobrepõe `clone` porque não basta copiar o objeto `Diretorio` para outro lugar. É necessário também copiar o objeto Vector. Se isto não for feito, o novo diretório vira um *mero atalho* do diretório anterior, pois a variável lista é uma referência e precisa ter seus dados copiados também.



Para testar todas as classes acima, usamos um programa que cria alguns arquivos, alguns diretórios e pede aos objetos que informem o seu caminho e o seu conteúdo, se forem diretórios. Esta classe tem apenas um método `main`.


```
/**
 * Esta classe é um aplicativo que usa os objetos
 * Diretorio e Arquivo para simular um sistema de
 * Arquivos.
 */

import sa.*; // é necessário importar porque esta classe
             // não pertence ao pacote sa.

public class SimSA {

   public static void main (String[] args) {

      ///// Objetos da simulação /////
      Diretorio root, java, awt, lang, event;
      Arquivo frameJ, frameC, buttonJ, stringJ,
              threadJ, actionT, mouseJ;

      /* criando diretórios e arquivos */
      root = new Diretorio ("/", null);   // root não tem pai
      java = new Diretorio ("java", root);
      awt = new Diretorio ("awt", java);
      lang = new Diretorio ("lang", java);
      event = new Diretorio ("event", awt);

      frameJ = new Arquivo("Frame", "java", awt, 1200);
      frameC = new Arquivo("Frame", "class", awt, 4700);
      buttonJ = new Arquivo("Button", "java", awt, 970);
      stringJ = new Arquivo("String", "java", lang, 11902);
      threadJ = new Arquivo("Thread", "java", lang, 7760);
      actionT = new Arquivo("ActionEvent", "txt", event, 3140);
      mouseJ = new Arquivo("MouseAdapter", "java", event, 5790);

      System.out.println("\nDIR /:\n");
      Arquivo[] list3 = root.dir();
      for (int i = 0; i < list3.length; i++) {
         System.out.println(list3[i]);
      }

      System.out.println("\nDIR /java:\n");
      Arquivo[] list4 = java.dir();
      for (int i = 0; i < list4.length; i++) {
         System.out.println(list4[i]);
      }
      System.out.println("\nDIR /java/awt:\n");
      Arquivo[] list1 = awt.dir();
      for (int i = 0; i < list1.length; i++) {
         System.out.println(list1[i]);
      }

      System.out.println("\nDIR /java/awt/*.java:\n");
```



```
        Arquivo[] list2 = awt.dir("java");
        for (int i = 0; i < list2.length; i++) {
           System.out.println(list2[i]);
        }

        System.out.println("\nCAMINHOS RECURSIVOS:");
        System.out.println("actionT.caminho(): " + actionT.caminho());
        System.out.println("threadJ.caminho(): " + threadJ.caminho());
        System.out.println("event.caminho(): " + event.caminho());

    }
}
```

Depois de compiladas todas as classes, podemos testar o sistema rodando o programa acima. Obteremos o seguinte resultado:

```
E:\Programas\> java SimSA

DIR /:

[1] java/       (1024) 2 arquivos

DIR /java:

[2] awt/        (1024) 4 arquivos
[3] lang/       (1024) 2 arquivos

DIR /java/awt:

[4] event/      (1024) 2 arquivos
[5] Frame.java  (1536)
[6] Frame.class (5120)
[7] Button.java (1024)

DIR /java/awt/*.java:

[5] Frame.java  (1536)
[7] Button.java (1024)

CAMINHOS RECURSIVOS:
actionT.caminho(): /java/awt/event/ActionEvent.txt
threadJ.caminho(): /java/lang/Thread.java
event.caminho(): /java/awt/event/

E:\Programas\>
```

Para criar estas classes foi preciso saber como era a estrutura dos métodos `clone`, `equals` e `toString`, que sobrepomos de `java.lang.Object`. Estas informações estão disponíveis na documentação da API. Aponte seu browser para a URL local `file:///c|jdk1.2/docs/`. Uma outra fonte de informações são os códigos-fonte, que estão disponíveis para todas as classes. Veja em `c:\jdk1.2\src\`.

Exercícios

1. Construa uma classe `Astro`, que representa um corpo celeste. Todo astro tem um nome e um outro astro ao redor do qual orbita. Crie também uma classe `Estrela`, que é um astro, e tem uma magnitude e uma cor. Sobreponha o método `Object.toString()` em `Astro` que imprima seu nome e a mesma informação sobre o astro que orbita. Sobreponha este método em `Estrela` para que imprima também a magnitude e a cor.
2. Teste o programa acima em uma classe `SistemaSolar`. Você deve criar objetos `sol`, `mercurio`, `venus`, `terra`, `marte`, `lua`, `phobos` e `deimos` (luas de Marte). Use `toString` para imprimir os resultados.
3. Repita o exercício anterior usando `SistemaAlphaCentauri`. Invente uns 5 planetas extras e acrescente mais duas estrelas, que devem orbitar a maior (Alpha Centauri é um sistema triplo). Pode haver mais planetas orbitando cada estrela, luas orbitando luas, etc.
4. Sobreponha `Object.clone()` em `Astro`. Garanta que quando um astro for clonado, ele continue orbitando o mesmo astro.

Classes, métodos e variáveis finais

Algumas classes não podem ter subclasses. São as classes declaradas com o modificador `final`. Se você tentar estender uma delas, o compilador acusará um erro. Declare suas classes como finais se você não deseja que seu comportamento seja alterado por subclasses. Classes consideradas "prontas", que ninguém deve estender, por decisão de projeto, devem ser declaradas `final`.

```
public final class Terminada { ... }
```

Você pode desejar apenas que um certo método não seja sobreposto pelas subclasses. Por exemplo, em um classe que faz verificação de senhas há um método chamado `valida()` que retorna `true`, se a senha for correta e false, se ela for false. Uma outra classe poderia estender esta classe e corrompê-la totalmente, sobrepondo `valida()` e fazendo-a sempre retornar `true`! Nesses casos, pode-se definir método como final, da forma:

```
public final valida(double senha) { ... }
```

Se um método for definido como `final` ou `static`, ele não pode sobreposto em uma subclasse. Em classes finais, todos os métodos são implicitamente finais.

Variáveis definidas como `final`, são consideradas constantes e não podem ter seus valores redefinidos, se forem tipos primitivos, ou não podem referir-se a outros objetos, se forem referências. Seus valores devem ser atribuídos no momento da criação e declaração:

```
pubic static final SQRT2 = 1.41421356
```

Classes e métodos abstratos

O que é um veículo? Alguém já viu um veículo na rua? Descreva um veículo: qual a sua cor, como se locomove?

Não é possível dar respostas claras e únicas a estas perguntas porque um veículo é um termo abstrato. Você até pode definir alguma coisa sobre um veículo. Ele deve ter uma capacidade de carga, deve ter uma velocidade. Se você implementar um modelo em Java, seu código até pode definir métodos que retornam estes valores, mas não pode definir com precisão outros que realizam coisas mais específicas como dar partida, virar à esquerda, etc. Veleiros, por exemplo, são veículos que viram à esquerda empurrando o leme para a direita!

Java permite que sejam criadas classes não totalmente especificadas, e você não precisa preencher o corpo dos métodos que fazem coisas abstratas. Basta declará-los como `abstract`, que eles não precisam ter corpo algum:

```
abstract void sigaParaEsquerda(int quantidade);
```

Métodos declarados como `abstract` não têm corpo. Eles devem terminar em ponto-e-vírgula. A conseqüência é que não se pode mais criar objetos com esta classe, já que parte dela não está especificada. Sendo assim, é preciso declarar toda a classe também `abstract`, sinalizando que ela não está completamente especificada.

```
public abstract class Veiculo { ... }
```

Na seção anterior vimos classes que não podiam ser estendidas. Com as classes abstratas ocorre o contrário. Elas *precisam* ser estendidas. E a classe que as estende precisa implementar (sobrepor) *todos* os métodos não implementados. Se isto não for feito, a subclasse também terá que ser declarada `abstract`.

Para que servem então as classes abstratas? Observe que as classes abstratas são classes e assim definem um *tipo de dados*. Elas só não podem ser construídas como objetos, mas seus métodos estáticos podem ser invocados e elas podem ser usadas para declarar tipos abstratos. Isto é útil, por exemplo, quando se constrói

um método que recebe um argumento e não se sabe exatamente o que ele vai receber em tempo de execução.

Suponha que tenhamos uma classe `Estacionamento`. Esta classe tem um método `admite()` que recebe um carro como argumento. Não seria muito interessante ter um método para cada tipo de carro existente, mesmo que fossem poucos, pois no dia seguinte em que o programa fosse rodar em um estacionamento, já poderia haver um novo modelo não suportado. Para resolver isto, usamos a classe abstrata `Carro`:

```
public abstract class Carro {
    public void estaciona();
}
```

Na nossa classe estacionamento, o método `admite` pode receber qualquer objeto do tipo Carro:

```
public boolean admite(Carro c) {
 c.estaciona();
}
```

Então, durante a operação do estacionamento, aparece um `Fusca fuscaBUM3456`, que é uma extensão de `Carro`. Ele é aceito sem problemas:

```
myPark.admite(fuscaBUM3456); // myPark é instância de Estacionamento
```

Depois, uma `FerrariF40 f40BZX2934` e uma `Charrete ch09`, ambos extensões de Carro:

```
myPark.admite(f40BZX2934);
myPark.admite(ch09);
```

Em cada classe `Fusca`, `Charrete` e `FerrariF40` o método `estaciona` deve estar implementado. Cada qual com suas particularidades, de forma que seja possível receber qualquer um desses veículos no estacionamento.

Interfaces

Herança múltipla pode ser uma característica desejável. Se você tem duas classes: `Gambá` e `Pato`, e quer fazer um `Ornitorrinco`, poderia ganhar tempo aproveitando algumas características dos dois bichos, não é? Mas Java não suporta a herança de múltiplas classes por ser uma fonte de muito código obscuro que pode levar a erros difíceis de detectar. A linguagem preservou,

porém, uma solução que mantém as boas qualidades da herança múltipla e evita suas desvantagens. São as *interfaces*.

Interfaces são uma espécie de classes abstratas onde *todos* os métodos são abstratos. Interfaces também não têm métodos estáticos nem variáveis. Além de declarações vazias de métodos pode haver apenas valores constantes.

A principal diferença de sintaxe entre interfaces e classes abstratas está na declaração. Uma interface deve ser declarada `interface`, e não `abstract class`. Uma interface também pode estender outras interfaces, mas não pode estender classe alguma. As declarações a seguir são válidas para interfaces.

```
interface Cockpit { ...}

public interface Painel { ... }

interface PaineldeCessna extends Painel { ... }
```

Os métodos abstratos declarados nas interfaces não precisam ter o modificador `abstract`.

Interfaces também definem *tipos de dados* mas não podem ser usadas para criar objetos. Para usar uma interface, uma outra classe deve *implementar* todos os seus métodos. Isto não é feito através da palavra-chave `extends`, mas usando `implements`.

```
public class Cessna implements Cockpit { ... }
```

A classe que implementa uma interface passa a ter uma relação "*é um*" com aquela interface, ou seja, o tipo da interface pode ser usado para representar objetos criados com aquela classe. Além disso, diferentemente da cláusula `extends`, `implements` pode receber mais de uma interface como argumento. As interfaces devem ser separadas por vírgulas. Uma classe pode ao mesmo tempo estender outra classe e implementar múltiplas interfaces.

```
public class Cessna extends AviãoHelice
             implements PaineldeCessna, Cockpit { ... }
```

Neste exemplo, *todos* os métodos existentes nas interfaces `PaineldeCessna` e `Cockpit` devem ser implementados ou o compilador irá reclamar dizendo que `Cessna` deve ser declarada `abstract`, por não definir todos os seus métodos (qualquer classe parcialmente implementada deve ser declarada `abstract`).

O uso de interfaces não caracteriza herança, mas composição. Por exemplo, podemos ter vários métodos para acionar controles de vôo (elevador, leme,

ailerão) declaradas na interface `Cockpit` e vários outros métodos para interpretar informações de um avião (altímetro, horizonte artificial, bússola) na interface `PaineldeCessna`. A interface sozinha não tem implementação. A classe `Cessna`, ao implementar as duas, herda os *protótipos* dos métodos. Qualquer lugar que recebe um `Cockpit`, recebe um Cessna, pois agora a referência para o `Cessna` é um `Cockpit` e um `Painel`. Qualquer `Piloto` que souber manusear um `Cockpit` e um `PaineldeCessna` conseguirá acionar os métodos de `Cessna` que controlam o avião. A implementação de todos os métodos da interface deve estar nas classes que a implementam. Se um `SimuladordeCessna` implementar as classes `Cockpit` e `PaineldeCessna`, o `Piloto` poderá interagir tão bem com ele como já fazia com o `Cessna`, pois o nome e formato dos métodos é o mesmo, mesmo que as implementações sejam diferentes.

Exemplos! Suponha que a classe `PaineldeCessna` fosse o seguinte:

```
interface PaineldeCessna {
 int altimetro();
 int horizonte();
 int bússola();
 int tacometro();
}
```

e que `Cockpit` fosse:

```
interface Cockpit {
 void partida();
 void leme(int posicao);
 void alieron(int posicao);
 void elevador(int posicao);
}
```

A declaração da classe `Cessna`, implementando as duas interfaces seria:

```
public class Cessna extends AviãoHelice
             implements PaineldeCessna, Cockpit {
...
   public int altimetro() { return altitude; }
   public int horizonte() { return horizonte; }
   public int bussola() { return direcao; }
      (...)
   public void aileron(int posicao) {
 // implementacao
   }
   public void elevador(int posicao) {
 // implementacao
```



```
    }
}
```

Se amanhã surgir um novo modelo de `Cessna`, o Piloto vai conseguir voá-lo tão bem quanto o antigo, desde que ele implemente as interfaces que já são familiares ao piloto. A interface é como o painel de um automóvel. Quem dirige um carro, dirige todos, já que todos têm direção, velocímetro, freio, etc. mesmo que a implementação de tais "métodos" seja diferente nos vários modelos.

As interfaces são, portanto, formas excelentes de *reutilizar* código e separar o *design* da implementação. Com ela, pode-se desenvolver programas extensíveis que permitem ter suas implementações alteradas no futuro sem precisar alterar dos componentes que já os utilizam.

Interfaces também são essenciais quando se precisa de herdar protótipos de mais de uma classe. São essenciais na criação de applets que precisam de funcionalidade adicional, como *multithreading*. Todo applet precisa estender a classe `java.lang.Applet` e para suportar *multithreading*, precisaria ser um `Thread` também. Como não há herança múltipla em Java, a única forma de implementar um objeto que cria *threads* a partir de si mesmo é através de interfaces.

O exemplo abaixo mostra como é possível utilizar uma interface para tornar um programa independente de forma de armazenamento. Inicialmente, temos uma interface chamada de `Armazenavel`:

```
public interface Armazenavel  {

    /** Escreve em qualquer lugar */
    public boolean save(String txt);

}
```

Esta interface é usada por um programa que pretende salvar um objeto do tipo `String`, em lugar ainda não definido. Pode ser em disco, em um banco de dados, em uma máquina remota, não interessa. Para o programa a seguir, ele está salvando em um objeto `Armazenavel`.

```
public class SalvaTextos implements Serializable {

    private transient Armazenavel peer;

    /** salva em disco */
    public boolean salvaString() {
```



```
        try {
            Armazenavel np = getStorage();
            if (np.save(String t)) {
                return true;
            } else return false;
        } catch (Exception e) {
            return false;
        }
    }

    /** retorna uma fonte de armazenamento */
    private Armazenavel getStorage() {
        // obtem uma instancia de uma fonte para armazenamento
        return peer;
    }
}
```

O método `getStorage` precisa obter um objeto que implemente `Armazenavel`. Ele pode conseguir isto de várias formas. Pode ser por escolha do usuário do programa ao escolher uma opção na interface do programa, por exemplo.

Para salvar em disco, é necessário ter uma implementação que lide com as peculiaridades da escrita em disco:

```
import java.io.*;
public class Disco implements Armazenavel, Serializable {
    (... variaveis ...)
    public boolean save(String t) {
    try {
            fos = new FileOutputStream("arquivo.txt");
            os = new ObjectOutputStream(fos);
            os.writeObject(t);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```

Se o `getStorage` receber um objeto `Disco`, ele salva em `Armazenável` e o texto é salvo em disco. Para salvar em um banco de dados, o `getStorage` precisaria retornar um objeto `BancoDados`, que lida com a escrita em um banco de dados:

```
import java.sql.*;
public class BancoDados implements Armazenavel {
```



```
(... variáveis e construtor ...)

    public boolean save(String t) {

        String sql = "INSERT INTO (...)";
        try {
            stmt.executeUpdate(sql);
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        return true;
    }

}
```

Conversão e Coerção

Na primeira seção deste módulo vimos que podíamos fazer conversões entre tipos primitivos através de atribuições, chamadas de métodos, promoção aritmética e coerção (*casting*). Tipos de referência também podem ser convertidos mas as regras são diferentes. Não é possível converter, através de uma simples atribuição, *cast* ou operação aritmética, um tipo de referência em um tipo de dados primitivo ou vice-versa. A única exceção ocorre com a classe `String`. Mas um tipo de referência pode ser convertido em outro tipo de referência, desde que haja alguma relação (extensão ou implementação) entre eles.

As regras de conversão entre tipos de referências são complexas porque há dois níveis a considerar. Como uma referência só vai encontrar realmente seu objeto em tempo de execução, é preciso considerar as conversões feitas nas duas situações. Um tipo sempre pode ser convertido em outro implicitamente (sem a necessidade de *cast*), se esta conversão for na direção da raiz da árvore de herança. Ou seja, uma subclasse sempre pode ser passada no lugar de uma superclasse. Por exemplo, qualquer classe pode ser usada onde se aceita `Object`. O contrário não é permitido sem uma conversão explícita. Exemplificando:

```
// AirCarrier extends Ship
AirCarrier air = new AirCarrier();
Ship s = new AirCarrier();
s = air; // OK!
air = s; // Illegal!
```

Uma referência do tipo de uma superclasse pode estar sendo usada em tempo de execução para apontar para um objeto do tipo de uma determinada subclasse. Neste caso, em tempo de execução, seria válido passar a referência

deste objeto (superclasse) para a referência da subclasse. Isto porque o objeto na memória *é* do tipo da subclasse. Mas o compilador não sabe disto e ele só permite a conversão se o programador se assumir o risco usando uma conversão explícita (*cast*). Mais exemplos:

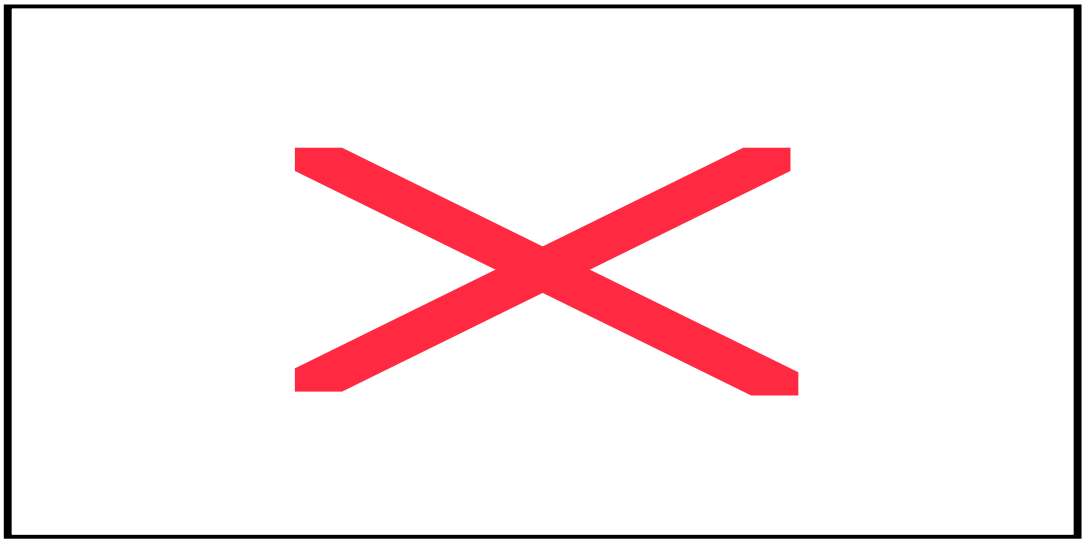
```
s = air; // Sempre OK!
air = (AirCarrier)s; // OK para o compilador!
```

O compilador sempre aceitará *casts* entre classes relacionadas. Mas pode ser que, na hora da execução, a classe não seja aquela prometida pelo programador. Neste caso, o programa lançará uma exceção.

```
Ship s = new Cruiser();
air = (AirCarrier)s; // OK para o compilador! Erro na execução!
```

No exemplo acima, como `s` não é um `AirCarrier` como esperado, haverá um erro em tempo de execução. O compilador acredita no programador e não acusa erro algum.

Toda a conversão sempre ocorre com referências. O objeto na memória nunca é afetado. É possível sempre ter várias referências para um mesmo objeto:



Às vezes o *cast* é usado simplesmente para permitir que o compilador saiba da existência de um campo ou método de um objeto. Se a sua classe `Texto` têm um método `imprime()` e um objeto do tipo `Texto` é passado para um método que aceita `Object` como argumento, a referência do tipo `Object` não vai

conseguir enxergar seu método. Para que isto seja possível, é necessário informar ao compilador que aquele `Object` é na verdade um `Texto`:

```
public void processa(Object o) {
    o.imprime(); // Erro de compilação! Compilador não conhece
                 // método imprime em Object!

    ((Texto)o).imprime(); // OK! Compilador agora sabe que
                          // objeto é um Texto!
```

`Object` pode ser usado para receber qualquer objeto, inclusive vetores. Vetores só podem ser convertidos em outros vetores ou em `Object` através de *cast* (desde que em tempo de execução, a referência de `Object` se refira a um vetor). Qualquer interface pode ser convertida em qualquer classe e vice-versa desde que usando *casts*.

JAVA 4

Java 2 API, Strings e Exceções

OS PRINCIPAIS PACOTES QUE COMPÕEM A API JAVA são apresentados neste módulo. Os módulos são apresentados de maneira superficial. Em detalhes serão mostrados exemplos de classes encapsuladoras, as classes String e StringBuffer e as classes usadas no controle de exceções.

Tópicos abordados neste módulo

- API Java
- Principais classes
- Strings
- Exceções

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- Identificar os principais pacotes da API Java
- Saber quando usar String e quando usar StringBuffer
- Conhecer a sintaxe das exceções
- Criar novas classes de Exceções
- Usar exceções para controlar condições excepcionais tempo de execução.

1.1. Java 2 Application Programming Interface

Java organiza as classes da sua biblioteca padrão em pacotes. Este conjunto de pacotes com suas classes forma a interface para a programação de aplicações, ou *API – Application Programming Interface*. Esse conjunto padrão é freqüentemente chamado de Core API, ou API de núcleo, porque faz parte dos ambientes de execução e programação Java em todas as plataformas que suportam a versão 2. Várias outras APIs existem que estendem a linguagem Java. Desde APIs proprietárias, como a IFC da Netscape ou a AFC, da Microsoft, como APIs da própria Sun como Java Foundation Classes (JFC), Java Multimedia API, Java Management API, Java Card, etc.

Os principais pacotes da API Java 2 estão descritos a seguir. São usados para diversos tipos de tarefas. Vários outros pacotes e subpacotes menos usados não estão descritos aqui.

Java oferece suporte à computação distribuída através de pacotes especiais, alguns dos quais não são distribuídos originalmente com o Java 2, como o pacote usado no desenvolvimento de servlets (javax.servlet). Para a construção de aplicações de rede, diversas classes e interfaces permitem a realização de operações que incluem desde o tratamento em baixo nível do fluxo de dados e de caracteres à operações complexas como a comunicação com objetos remotos, escritos em outra linguagem, de maneira totalmente transparente.

Os pacotes que começam com java.* são chamados de pacotes do núcleo ("core"). Os que começam com javax são extensões padrão ao núcleo ("core extensions"). Os pacotes sun.* e omg.* fazem parte de toda distribuição Java mas não são considerados da API Java fundamental. O pacote sun.* contém as ferramentas padrão da Sun como o compilador, o driver JDBC-ODBC, etc. e o pacote omg.* contém o suporte a CORBA.

java.applet

Contém a classe Applet e outras classes pouco usadas (são usadas internamente) como AppletStub e AppletContext. Contém uma classe AudioClip que representa um clipe musical.

java.awt

É o pacote que contém classes para o desenvolvimento de interfaces gráficas. Contém classes para lidar com a apresentação de informações em uma GUI, criação e posicionamento de componentes, tratamento de eventos, impressão e atalhos de teclado.

java.awt.image

Contém filtros e outras classes de manipulação de imagens.

java.awt.peer

Contém interfaces para um conjunto de ferramentas gráficas independentes de plataforma. As classes deste pacote raramente são usadas a não ser por classes do AWT ou extensões.

java.awt.datatransfer

As classes deste pacote definem uma estrutura genérica para a transferência de dados entre aplicações como classes que suportam um modelo de dados cut-and-paste de uma área de transferência.

java.awt.event

Classes e interfaces que suportam o modelo de eventos baseado em delegação. São três as categorias das classes e interfaces: classes que representam eventos da GUI; interfaces que escutam eventos (definem métodos que devem ser implementados pelos objetos interessados em serem notificados na ocorrência de um evento; e implementações triviais das interfaces de escuta (classes adaptadoras de eventos).

java.beans

Contém classes e interfaces para criar e usar componentes reutilizáveis chamados beans. Podem ser usadas de três formas: para criar ferramentas de construção de aplicações (para criar aplicações quase sem programar); para desenvolver novos beans para serem usados nesses construtores de aplicações e; para desenvolver aplicações que usam beans.

java.lang

Contém as classes fundamentais da linguagem Java como Object, String, Thread, System e Exception. Também contém classes empacotadoras de tipos primitivos (Byte, Short, Integer, Long, Float, Double, Character, Boolean e Void), para que os mesmos possam ser tratados como objetos, quando necessário.

java.lang.reflect

Contém classes que permitem que um programa em Java examine a estrutura de classes e obtenha informações completas sobre qualquer objeto, vetor, método, construtor ou campo de dados. É usado na programação genérica.

java.math

Contém duas classes apenas que oferecem suporte a operações aritméticas sobre inteiros de tamanho arbitrário e números de ponto-flutuante de precisão arbitrária.

java.io

Este pacote oferece suporte a operações e entrada e saída. Suas classes e interfaces podem ser classificadas em três grupos. O primeiro, que consiste da maior parte do pacote, é utilizado na construção e filtragem de fluxos (*streams*) de dados. Contém classes e interfaces que representam fluxos de entrada e saída de *bytes*, caracteres e objetos. O segundo grupo contém classes e interfaces que suportam a *serialização* de objetos (conversão de objetos em fluxos de *bytes* registrados com número de série para possibilitar o armazenamento persistente). O último grupo é usado para representar um sistema de arquivos.

java.net

Este pacote oferece suporte a aplicações de rede.. Contém classes e interfaces que permitem a implementação de clientes, servidores e protocolos TCP/IP utilizando *sockets*, datagramas UDP, endereços Internet, conexões HTTP e URLs.

java.rmi

Suporta a arquitetura de objetos remotos RMI – *Remote Method Invocation*, que permite o desenvolvimento de aplicações que invocam métodos em objetos localizados em máquinas virtuais diferentes. Baseia-se no protocolo JRMP – Java Remote Method Protocol que permite a comunicação entre máquinas virtuais Java.

java.sql

Pacote que suporta JDBC – *Java Database Connectivity*. Com estas classes e interfaces é possível desenvolver programas em Java que se comunicam com qualquer banco de dados relacional que suporte as operações mínimas do SQL92. Também oferece suporte ao desenvolvimento de *drivers* JDBC.

javax.servlet.

Extensão de núcleo (core extension) da API Java que oferece suporte ao desenvolvimento de componentes de servidor – os *servlets*. Servlets são componentes que rodam dentro de uma aplicação de servidor. Servlets HTTP podem ser usados como alternativa eficiente, aberta e independente de plataforma à tecnologias atualmente utilizadas nos servidores HTTP como CGI[1], ASP[2], ISAPI[3] ou Cold Fusion[4].

org.omg.CORBA e org.omg.cosNaming.

Extensão padrão do Java 2 (standard extension). Estes pacotes oferecem classes, interfaces e subpacotes que permitem o desenvolvimento de aplicações Java que usam a tecnologia e objetos remotos CORBA.

---

[1] Common Gateway Interface. especificação da W3C (World Wide Web Consortium) para aplicações no servidor invocadas pelo browser.

[2] Active Server Pages. Tecnologia da Microsoft que oferece uma alternativa ao CGI através de scripts embutidos em páginas Web.

[3] Internet Server Application Programmer's Interface. Alternativa ao CGI através de módulos ou bibliotecas dinâmicas desenvolvidas usando uma interface de programação proprietária. ISAPI é suportado pelos servidores Microsoft. Tecnologias concorrentes são NSAPI (servidor Netscape) e Apache Server API.

[4] Alternativa à tecnologia CGI (proprietária). Desenvolvida pela Allaire Inc.

javax.rmi

Extensão de núcleo (core extension) da API Java que oferece suporte a RMI usando o protocolo IIOP (Internet Inter-ORB Protocol) e o serviço de nomes da plataforma Java permitindo que objetos RMI se comuniquem com objetos CORBA.

java.naming

Pacote que oferece suporte a serviços de nomes e diretório. Utilizado neste trabalho para registrar nomes de objetos RMI quando utilizado via IIOP.

java.security

Contém classes e interfaces que representam certificados, chaves, assinaturas, etc. É necessário desenvolver ou adquirir as implementações que não são incluídas.

java.security.acl

Interfaces de alto nível para a manipulação de listas de controle de acesso.

java.security.interfaces

Interfaces básicas usadas pelo restante da API de segurança.

java.sql

Classes e interfaces que representam conexões, declarações, conjuntos de resultados e drivers usados em um acesso a banco de dados relacional.

java.text

Classes e interfaces usadas para internacionalização: formação de datas, representações decimais, mensagens textuais de acordo com parâmetros de um determinado local. Permite o desenvolvimento de aplicações internacionais.

java.util.zip

Classes que computam somas de verificação em fluxos de dados, realizam compressão e arquivamento de fluxos de dados nos formatos ZIP e GZIP.

1.2. Como usar os pacotes da API

Todo programa importa automaticamente o caminho para as classes que compõem a biblioteca `java.lang` (obs: é necessário importar `java.lang.reflect` se precisar ser usada). Para usar as demais classes dentro de um programa, devem ser chamadas pelo nome completo (por exemplo: `java.awt.Button`, `java.util.Date`) ou pelo nome abreviado desde que haja, no início do programa, uma declaração `import`, da forma:

```
import <classe usada>;
```

Sintaxes típicas do uso de `import` são:

```
import java.awt.Image;
import java.io.*;
```

O primeiro, permite que se use a classe `Image` dentro do programa (sem precisar usar `java.awt.Image` em todos os lugares). Já o segundo uso, permite que se use todas as classes do pacote `java.io` na forma abreviada.

O compilador Java usa a variável de ambiente `CLASSPATH` para descobrir onde procurar as classes da API Java. Ela combina a informação do `CLASSPATH` com a da declaração `import` para localizar uma classe. Por exemplo: `import java.awt.Frame` procura a classe `Frame` em

```
$CLASSPATH/java/awt/Frame.class
```

Se `CLASSPATH` for, por exemplo, `C:\jdk1.1.3\lib\classes`, o caminho absoluto para localizar a classe `Frame` será:

```
C:\java\lib\classes\java\awt\Frame.class
```

1.3. A Biblioteca Fundamental Java (java.lang)

A biblioteca mais importante de toda a API é `java.lang`. Todas as classes de `java.lang` são automaticamente importadas sem a necessidade de uma declaração import. A seguir estão relacionadas algumas das mais importantes classes deste pacote (consulte a documentação para maiores detalhes e outras classes):

Object

É a raiz de toda a hierarquia de classes em Java. Toda classe, existente ou não em Java, herda os parâmetros definidos em `Object`. É raro usar `Object` diretamente para criar instâncias. É mais comum usar `Object` para criar subclasses, quando não se estende outra classe, ou para usá-la como referência universal, como argumentos de métodos, construtores ou vetores.

Class

Classe que representa classes. Contém métodos para carregar classes dinamicamente e obter informações sobre elas. É muito usada em programação genérica.

Number, Character e Boolean

`Number` é uma classe abstrata que é superclasse de `Integer`, `Long`, `Byte`, `Short`, `Float` e `Double`, que juntamente com `Character`, `Void` e `Boolean` completam a coleção de classes *empacotadoras* de tipos primitivos. Como os tipos primitivos em Java não são objetos, eles se beneficiam destas classes quando precisam ser usados como objetos para passar valores por referência, serem incluídos em objetos `java.util.Hashtable`, `java.util.Vector` ou métodos que só recebem objetos.

`Integer` contém uma função (método estático) bastante usada para converter uma `String` em um inteiro: `int Integer.parseInt(String s)` recebe uma representação de número inteiro em formato `String` e devolve o valor inteiro. Os outros tipos não têm uma função assim e para fazer conversões é preciso *empacotá-los* em objetos para que se possa invocar métodos de conversão sobre eles. Abaixo estão alguns exemplos:

```
double d = new Double("12.783e-16").doubleValue();
float f = new Float("3.14159").floatValue();
long k = new Long("8516962").longValue();
boolean b = new Boolean("true").booleanValue();
char c = "B".charAt(0);
```

Math

É uma classe final que define constantes para os valores matemáticos π e e, além de definir um grande conjunto de funções matemáticas (métodos estáticos) para

a trigonometria, exponenciação e outras operações de ponto flutuante. Também contém métodos para calcular máximos e mínimos e gerar números pseudo-aleatórios. Exemplos:

```
double d = Math.sqrt(2);
int maior = Math(7, 9);
dado = (int)(Math.random() * 6)
```

System e Runtime

`System` define três variáveis estáticas que representam a entrada padrão (`in`), a saída padrão (`out`) e a saída padrão de erro (`err`). Além disso, define métodos que oferecem uma interface independente de plataforma para funções do sistema. Com `System` é possível, por exemplo, executar uma aplicação externa e controlar o processo resultante, recuperar as fontes do sistema, etc. `Runtime` encapsula várias funções do sistema que são dependentes de plataforma, como a coleta de lixo, e contém vários métodos que são chamados por `System`.

Process, Thread e Runnable

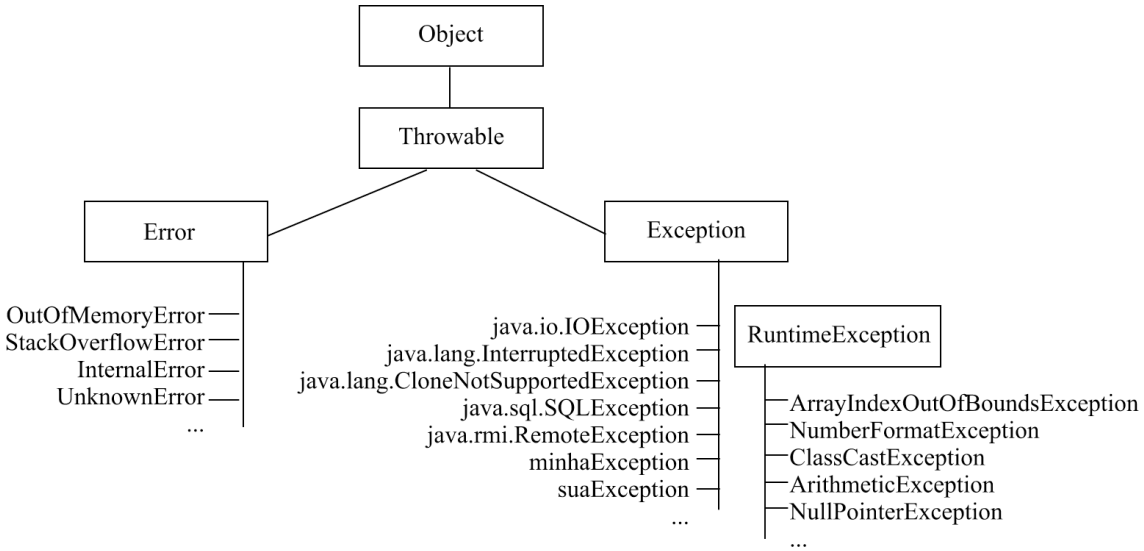
`Process` define uma interface independente de plataforma para processos que rodam externamente ao interpretador Java. `Thread` implementa suporte para múltiplas linhas de controle rodando no mesmo interpretador Java. Para criar uma linha de controle concorrente é preciso ou estender a classe `Thread` ou implementar a interface `Runnable` e passar o objeto resultante a um construtor do `Thread`. `Thread` define métodos para controle de prioridades, interrupção e agendamento de *threads*. `Runnable` declara um único método: `public void run()`, que deve ser implementado pelas classes que desejarem criar linhas de execução concorrentes.

Throwable, Error, Exception e suas subclasses

É a classe raiz da hierarquia de erros e exceções. Objetos `Throwable` são usados nas declarações `throw`, `catch` e `finally`. É pouco comum usar a classe `Throwable`. A classe `Error` é usada para definir erros dos quais não se espera recuperar, como por exemplo, a falta de memória. `Exception`, por sua vez, já serve para definir condições excepcionais que se espera ser possível corrigir sem abandonar a execução do programa, como, por exemplo, o fato de não conseguir encontrar um determinado arquivo.

A classe `Exception` ainda se divide em duas outras hierarquias importantes. A primeira são as classes derivadas diretamente de `Exception`, como as 7 exceções contidas no próprio pacote `java.lang`, várias outras definidas em outros pacotes, e as exceções definidas pelo usuário para representar condições excepcionais ocorridas em um programa que funciona corretamente (como uma conexão de rede indisponível, um arquivo não encontrado, uma URL digitada incorretamente, etc.). A segunda hierarquia é formada pelas classes derivadas de `RuntimeException`, que representa exceções que podem ocorrer durante o tempo de execução em um programa com *bugs*. As exceções deste tipo são avisos ao programador e ele deve corrigi-las.

A figura abaixo ilustra a hierarquia das exceções:



Cloneable

É uma interface que não contém método algum. Serve para sinalizar que o objeto criado pela classe que implementa esta interface pode ser copiado usando o método `Object.clone()`.

1.4. String e StringBuffer

São objetos que representam cadeias de caracteres. `String` é um tipo não-modificável e `StringBuffer` pode ter seu conteúdo alterado.

`Strings` são objetos especiais na linguagem Java. Por ser uma das classes mais usadas, possui formas diferentes de para instanciar objetos e certas

restrições para garantir sua integridade. É a única classe em Java utilizada para representar cadeias de caracteres.

Diferentemente de C, uma cadeia de caracteres em Java não é um vetor de `chars` terminado em `\u0000`, mas um objeto. Os objetos do tipo `String`, uma vez criados, não podem ser mais modificados, ou seja, não se pode acrescentar um caractere ou remover outro. Não há métodos públicos que permitem tais modificações. Para modificar o conteúdo de um `String`, é necessário passá-lo como argumento para uma outra classe que manipule caracteres, como a classe `StringBuffer` e depois criar um novo `String` com o conteúdo modificado.

Há vários métodos para operar com `Strings`, os mais usados são:

- `public char charAt(int index)`: retorna o caractere no local indicado por `index` (a primeira posição é 0).
- `public String concat(String str)`: concatena o `String` atual com outro, retornando um novo `String` (mesmo que "+").
- `public boolean endsWith(String sufixo)`: retorna `true` se o `String` termina com o sufixo passado como parâmetro
- `public boolean startsWith(String prefixo)`: retorna `true` se o `String` começa com o prefixo passado como parâmetro.
- `public int indexOf(String str, int inicio)`: retorna o índice do início do `String str`, a partir da posição início. Este método também pode receber como primeiro argumento um char e não ter o segundo argumento, na versão com `char` ou na versão com `String`.
- `public int lastIndexOf(String str, int fim)`: faz o mesmo que `indexOf` só que de trás para frente. Também tem as mesmas variações sobrecarregadas que o método anterior. O fim não faz parte do `String`, que vai de `inicio` a `fim-1`.
- `public int length()`: retorna o comprimento do `String` em caracteres.
- `public String replace(char antigo, char novo)`: troca o caractere `antigo` por um `novo` e devolve um novo `String`.
- `public String substring(int início, int fim)`: retorna um novo `String` iniciando na posição início do `String` atual e terminando em `fim-1`.

- `public String toLowerCase()` e `public String toUpperCase()` retornam um novo `String` em caixa-alta ou caixa-baixa respectivamente.

Observe que todos os métodos que fazem alterações devolvem um novo objeto. A classe `String` produz objetos imutáveis.

A classe `StringBuffer` possui métodos que modificam os objetos do seu tipo, como `insert(int posição, String str)`, `append(String str)`, `reverse()`, `setCharAt(int posição, char caracter)` e `setLength(int novoTamanho)`. `StringBuffers` devem ser convertidos em `Strings` (usando `toString()` ou passando como construtor de um novo `String`) para poderem ser usados.

A concatenação de `Strings` é uma operação extremamente ineficiente. Para realizar concatenações, o sistema cria vários objetos que logo são descartados. Se você realizar muitas concatenções, o uso de `StringBuffer` é recomendado (use `append()`) pois esta última classe não cria novos objetos na concatenação. No final, você sempre pode converter tudo em `String`.

`Strings` podem ser criados da mesma forma como se cria objetos comuns:

```
String s = new String("I am a String!");
```

ou da forma mais usual:

```
String s = "I am a String!";
```

Ambos produzem o mesmo resultado só que o último tem um tratamento especial e é tratado mais como um tipo básico que como um `String`. Se você comparar usando "==" dois `Strings` criados usando `new`, eles serão apontados como diferentes (a referência foi comparada), mas se os dois tiverem sido criados usando a forma de atribuição, o sistema os considera iguais. O sistema dá este tratamento especial a `Strings` porque eles são imutáveis.

Testes

1. (Roberts/Heller 97) Dado uma string construída usando `s = new String("xyzzy")`, quais das chamadas listadas abaixo modifica o string?
   A. `s.append("aaa");`
   B. `s.trim();`
   C. `s.substring(3);`
   D. `s.replace('z', 'a');`

E. `s.concat(s);`

1.5. Exceções

Quando acontece uma situação inesperada na execução de um programa, é necessário que exista uma rotina que lide com a situação e tente contorná-la se possível. Caso tal precaução não seja tomada, o programa deixará de funcionar como esperado, podendo fornecer resultados incorretos ou até abortar a execução.

Para desenvolver um código robusto, é necessário prever tais situações inesperadas e criar meios de contorná-las. Em todas as linguagens há maneiras de fazer isto. Pode ser tão simples quanto criar algumas expressões condicionais (por exemplo: testar se ocorrem divisões por zero) ou utilizando códigos de erro. Esta última alternativa é bastante usada em C e C++ e, apesar de aumentar a robustez do código, torna-o menos legível e mais complicado.

Para lidar com situações excepcionais Java proporciona uma forma limpa de verificar a ocorrência de erros e tratá-los se possível: as exceções.

O modelo de exceções já é adotado em outras linguagens, como o C++ (versões recentes) e Delphi. Em Java, seu uso é obrigatório em muitos casos como em programas que lidam com entrada e saída, conexões de rede, etc. A linguagem também permite que se defina novas exceções para lidar com situações particulares a um determinado programa.

Com as exceções, podemos programar sem preocupações com os erros que possam ocorrer em tempo de execução usando blocos `try-catch`. O trecho onde poderá haver erros é colocado dentro de um bloco `try { ... }` e depois dele, usamos um ou mais blocos `catch` para capturar exceções e fazer alguma coisa para lidar com elas.

Em muitos casos, o compilador *exige* que determinado bloco seja incluído em blocos `try... catch`, ou que seja declarado, na declaração do método, que tal exceção pode ocorrer. Isto é feito usando a palavra-chave `throws`:

```
public void m() throws Exception { ... }
```

Podemos também provocar uma exceção à força, usando `throw`:

```
try {
   (...)
   Exception e = new Exception();
   throw e;
```



```
    (...)
} catch (e) {
    /* ... */
}
```

Finalmente, podemos também agrupar várias coisas que devem ser feitas em caso de erro em um bloco `finally`. Se houver um erro, e o programa for abortar, antes de sair ele executa tudo o que for incluído no bloco `finally`.

A sintaxe básica para as exceções é a seguinte:

```
try { ... }
[catch (TipoDeExceção e1) {...}]
[catch (TipoDeExceção e2) {...}... ]
[finally { ... } ]
```

Pode haver zero ou mais blocos `catch`, para tratar exceções diferentes. O bloco `finally` também é opcional. Um `try`, porém, deve vir seguido de pelo menos um `catch` ou pelo `finally`.

Exceções são objetos. Quando acontece uma condição excepcional, o programa ou o sistema lança um objeto do tipo que representa a exceção ocorrida.

É fácil criar um programa que provoca exceções. Elas podem ser provocadas explicitamente (usando `throw`), de propósito, pelo programador para sinalizar uma condição excepcional, ou por métodos da API (métodos que declaram, com cláusula `throws`, que provocam exceções). As exceções abaixo estão entre as mais comuns:

- `ArrayIndexOutOfBoundsException` ocorre quando um vetor é acessado além dos seus limites inferior ou superior.
- `NumberFormatException` ocorre quando um `String` não pode ser convertido em número porque está no formato incorreto.

Ambas são subclasses de `Exception`, portanto, podem ser passadas para qualquer `catch` que pega `Exception`. Você sempre pode colocar um

```
catch (Exception e) { ... }
```

no final de todos os seus blocos `try {...}`. Isto irá pegar qualquer exceção, mas não irá pegar os erros (`Error`). Se você fizer isto, você não deve ter nenhum outro bloco `catch` com outra exceção, subclasse de `Exception`, depois deste senão o compilador reclamará dizendo que a declaração seguinte nunca será alcançada.

Isto acontecerá porque `Exception` a pegará primeiro. A ordem dos fatores importa.

Algumas outras exceções bastante comuns são:

- `NullPointerException` ocorre quando uma referência a um objeto ou vetor é usado sem ter um objeto correspondente ou em qualquer outro caso em que uma referência aponta para `null`.
- `SecurityException` é uma das mais comuns nos browsers. Ocorre quando um applet tenta violar as restrições de segurança do browser e tenta fazer algo ilegal como escrever em disco, por exemplo.
- `ArithmeticException` ocorre sempre que acontece uma divisão por zero em operações com inteiros. É a única exceção que ocorre nestes casos.

Lançamento de exceções

Para lançar uma nova exceção, basta criar um novo objeto do tipo desejado e lança-lo com a palavra-chave `throw`.

```
throw new SuaException();
```

Depois de lançada a exceção, ela deve ou ser tratada no local ou propagada para cima na hierarquia, isto é, o método onde ela ocorre simplesmente declara que ela pode ocorrer e os outros métodos, que usam a classe e chamam esse método é que terão que lidar com a exceção.

Por exemplo, você poderia definir, em `Círculo`, o método:

```
public void setRaio(double raio) {
    if (raio <= 0) {
        throw new Exception();
   }
}
```

Ao compilar tal classe, o compilador reclama dizendo que: ou você trata esta exceção ou declara que `setRaio throws Exception`. A palavra-chave `throws` é usada apenas em declarações. Não a confunda com `throw`! Então fazemos

```
public void setRaio(double raio) throws Exception {
    if (raio <= 0) {
        throw new Exception();
   }
}
```


e o programa compila sem problemas. Mas depois, ao compilar o programa `Desenha`, que cria novos círculos e define novos raios, temos a mesma mensagem de erro porque o nosso método está chamando `setRaio`. Desta vez, optamos por tratar a exceção:

```
public static void main (String args[]) {
    try {
        Circulo c = new Circulo();
        c.setRaio(-1);
    } catch (Exception e) {
        try {
            c.setRaio(1);
        } catch (Exception e) {}
    }
}
```

Criação de novas exceções

A exceção `Exception` no exemplo acima nada diz sobre o tipo de exceção que ocorreu. Neste caso, o mais interessante é ter uma exceção especial que caracterize o problema. Para criar novas classes de exceções, devemos estender a classe `Exception`:

```
class RaioNegativoException extends Exception {}
```

e isto é tudo o que é necessário fazer. Podemos definir outros construtores para poder passar mensagens, outros métodos, qualquer coisa que uma classe pode ter, uma Exceção pode ter, mas a sintaxe acima já resolve tudo. Agora declaramos:

```
public void setRaio(double raio) throws RaioNegativoException {
    if (raio <= 0) {
        throw new RaioNegativoException();
   }
}
```

E agora o programa que cria círculos pode saber quando acontece uma exceção deste tipo e tratar de acordo, usando um `catch` específico só para este tipo de exceção.

Testes

Os testes a seguir são do livro "Java 1.1 Certification Study Guide" de Roberts/Heller.

1. Considere a seguinte hierarquia de classes e fragmentos de código:

```
                    java.lang.Exception
                            \
                       java.io.IOException
                        /                \
java.io.StreamCorruptedException    java. net.MalformedURLException
```

```
 1. try {
 2.   URL u = new URL(s); // assume s is a previously defined String
 3.   Object o = in.readObject(); // in is a valid ObjectInputStream
 4.   System.out.println("Success");
 5. }
 6. catch (MalformedURLException e) {
 7.   System.out.println("Bad URL");
 8. }
 9. catch (StreamCorruptedException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Que linhas são impressas se os métodos das linhas 2 e 3 completam com sucesso sem provocar exceções?

A. Success
B. Bad URL
C. Bad File Contents
D. General Exception
E. doing finally part
F. Carrying on
G.

2. Considere a seguinte hierarquia de classes e fragmentos de código:

```
            java.lang.Throwable
              /              \
    java.lang.Error        java.lang.Exception
```



```
                /                       \
  java.lang.OutOfMemoryError    java.io.IOException
                                /                 \
     java.io.StreamCorruptedException    java.net.MalformedURLException
```

```
 1. try {
 2.   URL u = new URL(s); // assume s is a previously defined String
 3.   Object o = in.readObject(); // in is a valid ObjectInputStream
 4.   System.out.println("Success");
 5. }
 6. catch (MalformedURLException e) {
 7.   System.out.println("Bad URL");
 8. }
 9. catch (StreamCorruptedException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```

Que linhas são impressas que a linha 3 provoca um OutOfMemoryError?

A. Success
B. Bad URL
C. Bad File Contents
D. General Exception
E. doing finally part
F. Carrying on

3. Considere a seguinte hierarquia de classes e fragmentos de código:

```
                        java.lang.Exception
                                \
                          java.io.IOException
                           /                \
  java.io.StreamCorruptedException    java. net.MalformedURLException
```

```
1. try {
2.   URL u = new URL(s); // assume s is a previously defined String
3.   Object o = in.readObject(); // in is a valid ObjectInputStream
4.   System.out.println("Success");
5. }
6. catch (MalformedURLException e) {
```

```
 7.   System.out.println("Bad URL");
 8. }
 9. catch (StreamCorruptedException e) {
10.   System.out.println("Bad file contents");
11. }
12. catch (Exception e) {
13.   System.out.println("General exception");
14. }
15. finally {
16.   System.out.println("doing finally part");
17. }
18. System.out.println("Carrying on");
```


Que linhas são impressas se o método na linha 2 provoca uma MalformedURLException?

A. Success
B. Bad URL
C. Bad File Contents
D. General Exception
E. doing finally part

F. Carrying on

JAVA 5

Interface gráfica

ESTE MÓDULO APRESENTA OS CONCEITOS POR TRÁS DA PROGRAMAÇÃO DA INTERFACE GRÁFICA DO USUÁRIO (GUI) e apresenta a API Abstract Window Toolkit (AWT) – um conjunto de ferramentas independente de plataforma para o desenvolvimento de aplicações gráficas. É demonstrado o uso de gerentes de layout, da biblioteca de componentes e do tratamento de eventos.

Tópicos abordados neste módulo

Objetivos

No final deste módulo você deverá ser capaz de:

1.1. Programação da AWT

Java oferece uma biblioteca de objetos independentes de plataforma para a programação da Interface Gráfica (GUI). Esses objetos compõem o *Abstract Window Toolkit (AWT)*, que é um conjunto de ferramentas (classes, interfaces e métodos) abstratas para o desenvolvimento de aplicações que utilizam o sistema de janelas.

Hoje, com a grande base instalada de computadores com monitores gráficos de alta resolução, nenhum software que dependa de uma interface orientada a caractere consegue competir no mercado. O AWT oferece botões, caixas de texto, janelas, quadros, barras de rolamento, e todos os objetos e mecanismos

necessários para desenvolver aplicações gráficas, orientadas a eventos, em qualquer das plataformas que suportam o ambiente de execução (*runtime*) Java.

O nível de abstração do AWT é alto o suficiente para garantir a independência de plataforma. O AWT não determina a forma de apresentação das janelas; tarefa que fica com o sistema operacional nativo. Desta forma, um programa gráfico rodando no ambiente X-Window/ Motif usa os botões, barras de rolamento e caixas de rádio característicos da interface Motif, enquanto o mesmo programa rodando no Macintosh ou Windows95 irá se apresentar com os objetos de janelas característicos de seus respectivos sistemas. Um outro pacote para desenvolvimento de aplicações gráficas – o JFC (*Java Foundation Classes*) ou *Swing*, pode ser usado para desenvolver interfaces gráficas mais sofisticadas, independentes dos recursos oferecidos por cada sistema operacional. Aplicações Swing podem ou não ter o *look&feel* de aplicações nativas.

Neste capítulo, estudaremos as classes mais importantes do pacote `java.awt`.

Lógica da Programação Orientada a Eventos

A principal diferença entre os programas que rodam em ambientes de janelas e os programas que executam na linha de comando está na forma como são executados. A lógica de um é o inverso da outra. Em programas de linha de comando as ações ocorrem seqüencialmente do começo ao fim, podendo-se prever quando e onde o usuário terá opções de entrar com dados ou interferir na execução do programa.

Já um programa orientado a eventos tem uma lógica assíncrona. Não é possível prever qual botão o usuário irá apertar e quando. Em vez de um fluxo contínuo de controle, o programa fica esperando por uma ação do usuário. Para isso, ele tem que ficar monitorando os eventos que ocorrem no sistema operacional, como movimento do mouse e teclas digitadas, para saber se ele precisa realizar alguma ação baseado neles.

Quando o usuário realiza alguma ação (movendo o mouse sobre uma janela, por exemplo), o sistema de janelas captura o evento e o passa para uma rotina que foi criada para manuseá-lo. A rotina deverá lidar com o evento e com a ação que está associada a ele. Por exemplo, se um mouse é clicado sobre o botão ☒ no canto superior direito de um programa Windows95, a rotina deve tomar

conhecimento que esse evento ocorreu e providenciar a execução de uma resposta ao evento: geralmente o fechamento da janela ou a chamada de uma rotina mais complexa para lançar uma janela de diálogo para confirmar a saída.

Em Java, todos os eventos do sistema operacional podem ser capturados e manuseados através das subclasses de `java.awt.AWTEvent`. O tratamento de eventos é realizado por um objeto especialmente criado para este fim. Este objeto deve implementar os métodos de manuseio de eventos para o tipo de evento desejado. A classe que produz os eventos cadastra objetos que desejam ser informados sobre o evento que irá acontecer. O exemplo abaixo mostra uma aplicação onde o aperto de um botão provoca o fechamento do programa.

```
// classe que trata evento de fecha janela
class WindowCloser extends WindowAdapter {
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}  // end class


// classe (janela) onde que produz o evento
(...)
this.addWindowListener(this.new WindowCloser());
(...)
```

Quando o botão de fechar janela é apertado, um objeto *WindowCloser* é criado e seu método `windowClosing()` é executado, encerrando o programa. Outras respostas poderiam ocorrer ao mesmo evento, bastava cadastrar novas classes manuseadores de eventos usando `addWindowListener()`. Eventos de ação (apertar botão, selecionar menu) são sempre tratados como `Action` events. Outros tipos de eventos são `ItemEvent`, `MouseEvent`, etc. Cada qual tem seus próprios ouvintes (*listeners*) e métodos.

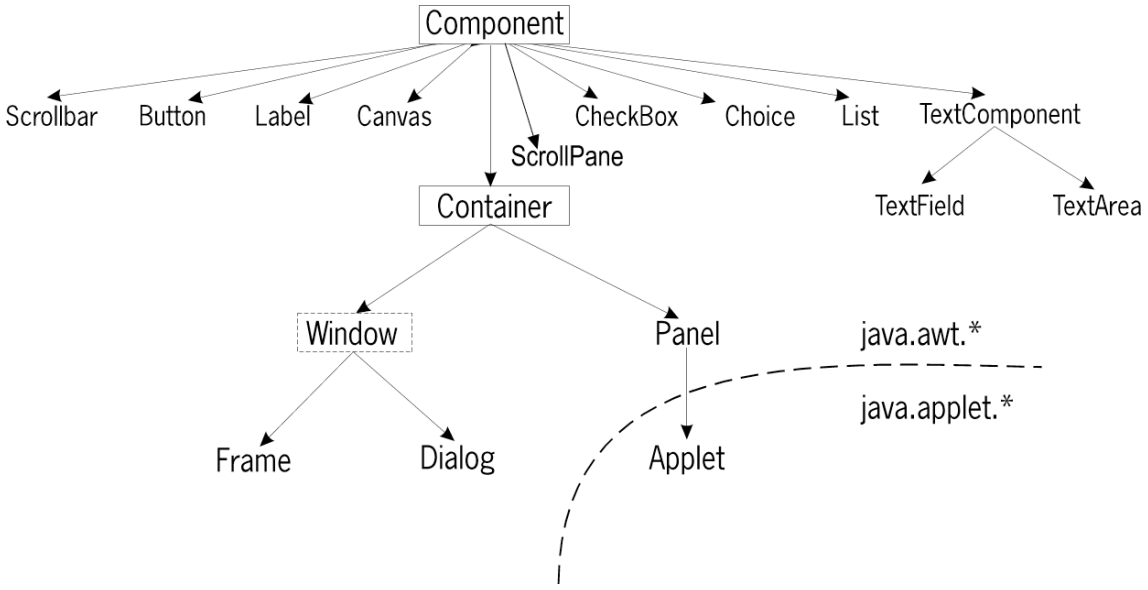
Hierarquia e Funções do AWT

O pacote `java.awt`, que é o *Abstract Window Toolkit*, pode ser dividido em quatro categorias:

- **Subclasses de Component e MenuComponent:** um conjunto de objetos da GUI: "widgets" (menus, botões, caixas de escolha, caixas de rádio, etc.) e recipientes (janelas, quadros, painéis, caixas de diálogo).

- **Sistema de manuseio de eventos da AWT:** classe `AWTEvent` e suas subclasses (`pacote java.awt.event`) e métodos de captura de eventos.
- **Gerentes de layout:** organizam os "widgets" em um "container" de acordo com uma estrutura definida. (implementações da interface `LayoutManager`).
- **Classes de suporte a operações gráficas:** diversas classes que permitem desenhar um polígono, preencher uma elipse, mudar uma cor, mostrar uma imagem, etc.

A figura abaixo ilustra a hierarquia dos componentes da GUI que são subclasses de `Component` ("widgets" e "containers").



Componentes e Recipientes (Containers)

Como ilustra a figura acima, `Containers` são uma sub-classe dos `Components`, assim como todos os widgets. Os widgets, que são componentes, podem ser colocados em recipientes (`containers`), que também são componentes e por sua vez podem ser colocados em outros recipientes. A diferença entre os dois é ilustrado abaixo:

Components	Containers (são Components)
1. São exibidos na tela mostrando todo o seu	1. Recebem e organizam widgets de acordo

conteúdo.	com um controle de layout.
2. Capturam e lidam com eventos relacionados aos widgets, teclado e mouse.	2. Sobrepõem métodos de um componente, para captura e manuseio de eventos.

Component

É a superclasse de todos os objetos da GUI (exceto menus). Têm uma posição (x, y) e uma dimensão (altura, largura), e podem ser desenhados na tela. Não é possível instanciar um objeto `Component` diretamente (ele não tem método construtor), somente uma das suas subclasses. `Component` define vários métodos, todos herdados pelas suas subclasses. O método mais importante de `Component` é:

```
public void paint(Graphics g)
```

Este método, quando chamado pelo sistema de tempo de execução, desenha o componente e todo o seu conteúdo na tela.

Além deste método, `Component` ainda possui dezenas de outros que alteram sua posição e tamanho (`setBounds`, `setSize`), e alteram propriedades gráficas.

Container

Implementa um componente que pode conter outros componentes e tratá-los como um grupo. Também não pode ser instanciado diretamente. Você deve usar uma das suas subclasses `Panel`, `Frame` ou `Dialog` (subclasses de `Window`). O principal método de `Container`, herdado por todas as suas subclasses, é:

```
public Component add(Component c)
public Component add("North", Component c)
```

que adiciona um componente ao `Container`. A maioria dos outros métodos são relacionados com gerentes de layout.

Window

É uma janela em branco, sem bordas ou barra de menu. É um recipiente cujo layout default é `BorderLayout` (veja adiante). Pode ser instanciada, mas em geral não é usada diretamente. Utiliza-se muito as suas subclasses `Frame` e `Dialog`.

Frame

É uma especialização de `Window`, que tem uma borda, pode ter um título, uma barra de menu, um ícone e um cursor. Várias constantes definidas nesta classe

definem diversos tipos de cursor. `Frame` geralmente é a base de uma aplicação GUI. É o recipiente que recebe todos os componentes que formam a interface gráfica do programa.

Há duas formas de se criar um objeto `Frame`:

```
Frame f = new Frame()
Frame f = new Frame("Título da Janela");
```

A primeira forma cria um quadro sem título. A segunda cria um quadro com o título "Título da Janela" na sua barra de título. Também, pode-se usar:

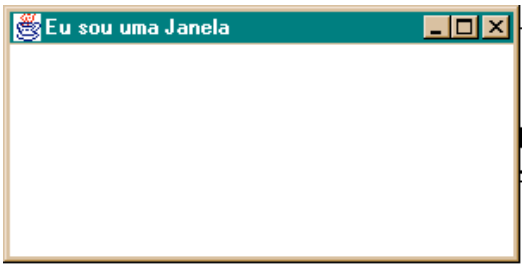
```
f.setTitle("Título da Janela");
```

para definir o título. Outros métodos são: `setCursor(int tipo_cursor)`, que especifica um cursor; `setMenuBar(MenuBar mb)`, que define uma barra de menu e `setIconImage(Image img)`, que define uma imagem para o ícone do quadro.

Uma maneira mais comum de se criar um `Frame` é fazê-lo a base de uma aplicação. **Exemplo:**

```
import java.awt.*;
public class Quadro extends Frame {

    Quadro() {
       super ("Eu sou uma Janela");
       setSize(300,150);
       setLocation(300,200);
       setVisible(true);
    }

    public static void main(String a[]) {
       new Quadro();
    }
}
```

Faz aparecer na tela a janela a seguir.

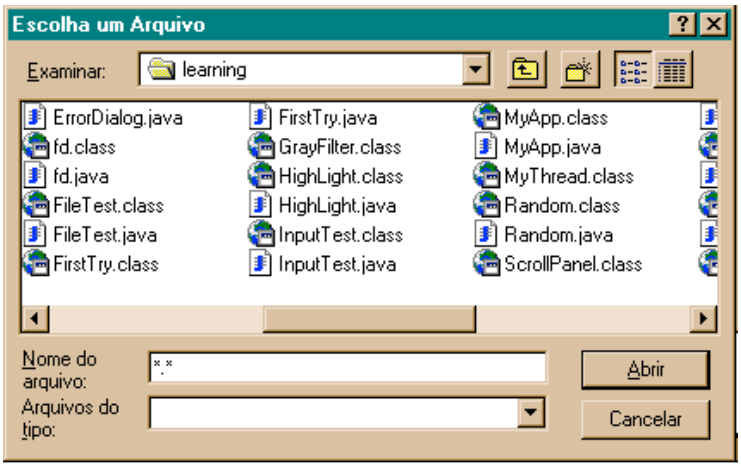


Dialog

É uma janela de caixa de diálogo. Aplicações são diálogos Sim/Não, OK/Cancela, informações, avisos, erros, perguntas, etc. Uma subclasse

importante é `FileDialog`, que faz aparecer a caixa de diálogo para seleção de um arquivo. Adicionando as linhas a seguir no `main` do programa anterior, faz aparecer uma janela `FileDialog`:

```
Quadro() {
    setBounds(300,150, 300,200);
    FileDialog fd = new FileDialog(this, "Escolha" +
                                   " um Arquivo");
    fd.setVisible(true);
    add(fd);
    setVisible(true);
}
```



Panel

É um recipiente (`Container`) que não cria uma janela própria (como faz `Dialog` e `Frame`). É útil para dividir partes de uma interface maior (como um `Frame` ou `Dialog`) ou mesmo outro `Panel`.

`Panel` quase não tem métodos próprios. Sua função básica é proporcionar um layout para a sua sub-classe `Applet` e como forma de dividir um layout complexo.

Applet

Applet é uma subclasse de `Panel` que faz parte do pacote `java.applet`. Applets são discutidos no final deste capítulo.

1.2. Eventos da AWT

Em Java, todo evento é uma subclasse de `java.util.EventObject`. Os eventos do AWT são subclasses de `java.awt.AWTEvent` e os diversos tipos básicos pré-definidos de eventos do AWT como `MouseEvent`, `WindowEvent`, `ActionEvent`, etc. são parte do novo `java.awt.event`.

Todo evento tem um objeto origem que é obtido através do seu método `getSource()`. Cada evento do AWT tem um tipo que é obtido com `getID()`. A identificação é usada para distinguir entre os diversos tipos de eventos que ocorrem em uma mesma classe (como `MouseEvent.MOUSE_CLICKED` e `MouseEvent.MOUSE_EXITED`). As classes contém os métodos que têm algo a ver com o tipo de evento ocorrido. Por exemplo, `MouseEvent` tem métodos `getX()`, `getY()`, etc.

O modelo de eventos baseia-se no conceito de um objeto de escuta (*listener*). Qualquer objeto interessado em receber eventos é um *listener*. Um objeto que gera eventos (*event source*) mantém uma lista de *listeners* interessados em serem notificados quando um determinado evento ocorrer. Eles também possuem métodos para que os *listeners* se cadastrem para receber eventos.

A notificação do *listener* pela fonte ocorre através de uma invocação de método do *listener*, passando para ele uma instância de `EventObject`. É necessário que todos os *listeners* implementem o método necessário para que possam ser avisados. Isto é feito através da implementação de uma ou mais interfaces apropriadas para o tipo de evento que se deseja receber. Os métodos implementados recebem um único argumento que é o objeto (instância de `EventObject`) que corresponde ao *listener*. O objeto deve conter toda a informação necessária para que um programa possa responder ao evento.

A tabela abaixo relaciona tipos de evento, métodos e interfaces de escuta (*listeners*) [Java In a NutShell. 2nd. Ed.]:

Classe	**Interface (listener)**	**Métodos (listener)**
`ActionEvent`	`ActionListener`	`actionPerformed()`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged()`

`ComponentEvent`	`ComponentListener`	`componentHidden()`
		`componentMoved()`
		`componentResized()`
		`componentShown()`

`ContainerEvent`	`ContainerListener`	`componentAdded()`

		componentRemoved()
FocusEvent	FocusListener	focusGained() focusLost()
ItemEvent	ItemListener	itemStateChanged()
KeyEvent	KeyListener	keyPressed() keyReleased() keyTyped()
MouseEvent	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()
	MouseMotionListener	mouseDragged() mouseMoved()
TextEvent	TextListener	textValueChanged()
WindowEvent	WindowListener	windowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowIconified() windowOpened()

Para ter acesso aos eventos e seus métodos, deve-se importar as classes de `java.awt.event.*`. Todos os métodos de eventos sempre têm a forma:

```
public void nome (XXXEvent evt) { ... }
```

onde `XXX` é o tipo de evento (coluna 1) tratado. Um evento só será encaminhado para um ouvinte se ele estiver cadastrado na fonte através de um método do tipo:

```
fonte.addXXXListener(ouvinte do tipo XXXListener);
```

onde fonte é o objeto-fonte onde ocorre o evento e `XXX` o tipo do evento (`Mouse`, `Action`, `Window`, etc.). O ouvinte do tipo `listener` é um objeto criado com a classe que implementa a interface correspondente ao tipo de evento tratado.
Por exemplo, para cadastrar um botão para que ele responda a eventos de clique do mouse sobre ele (eventos do tipo `Action`) é preciso fazer:

```
Button b = new Button("Botão"); //criação
b.addActionListener(new OuvinteAcao());
```

considerando-se que existe uma classe chamada `OuvinteAcao()` que implementa a interface `ActionListener` (e consequentemente o método `actionPerfor-`

`med()`). O método deverá descrever as ações a serem tomadas quando o evento ocorrer.

1.3. Componentes da AWT

Os componentes são geralmente são adicionados aos `Frames`, `Dialogs` ou `Panels`. Para criar um componente, é preciso declarar uma referência para ele e instanciá-lo. Em seguida, deve ser adicionado a um recipiente. Quando o recipiente se tornar visível, todos os componentes que ele tiver aparecerão na tela. O posicionamento e dimensionamento do componente geralmente é ditado pela política de *layout* do recipiente. Caso o recipiente não tenha política de *layout* (pode ter sido desligado com `setLayout(null)`), é preciso ainda definir as dimensões, margens e localização do componente.

Os passos para a criação e utilização dos componente são:

1. Declarar a referência (geralmente como membro do objeto que constrói a interface gráfica):

   ```
   private Button b1;
   ```

2. Instanciar o objeto.

   ```
   b1 = new Button("Enviar Dados");
   ```

3. Adicioná-lo ao recipiente (ou a um painel dentro do recipiente). A forma de colocar um componente dentro de outro depende do *layout* utilizado. Para o `FlowLayout`, poderia ser:

   ```
   this.add(b1);
   ```

4. Cadastrar o objeto com um manuseador (ouvinte) de eventos

   ```
   b1.addActionListener(new MenusEvt);
   ```

5. Definir um comando para identificar o evento

   ```
   b1.setActionCommand("env");
   ```

Para os itens 4 e 5 acima, estamos supondo a existência de uma classe `MenusEvt` que poderia ter a seguinte estrutura:

```
class MenusEvt implemens java.awt.event.ActionListener {
  public void actionPerformed(ActionEvent e) {
    String comando = e.getActionCommand();
    ...
```



```
    }
}
```

Scrollbar

Barra de rolamento. Exemplo de criação:

```
Scrollbar s =
    new Scrollbar(Scrollbar.VERTICAL, 50, 10, 0, 99);
```

Os argumentos são respectivamente: a orientação da barra (`Scrollbar.HORIZONTAL` ou `Scrollbar.VERTICAL`), a posição inicial da barra deslizante, o tamanho da parte visível da área e os valores do início e fim da escala. Há vários métodos para controlar a posição das barras. Para capturar o valor da posição selecionada, deve-se criar um manuseador de eventos que implemente a interface `AdjustmentListener` (evento `AdjustmentEvent`) e cadastrar o `Scrollbar` com `addAdjustmentListener()`.

Button

Botão. Exemplo de criação:

```
Button b1 = new Button("Aperte-me");
```

Para capturar o evento, implemente um `ActionListener`. Para cadastrá-lo, use `addActionListener()`. Veja exemplo na seção anterior.

Canvas

É um componente que não cria um desenho *default* ou qualquer mecanismo próprio de manuseio de eventos. É uma tela que serve para desenhar gráficos ou receber entrada do usuário. `Canvas` normalmente é utilizada através de uma subclasse para sobrepor o seu método `paint` e obter um contexto gráfico onde se possa usar os métodos de `Graphics` (para desenhar retângulos, elipses, polígonos, imagens, etc.).

O exemplo abaixo cria um objeto `Canvas` e o adiciona ao contexto atual:

```
Canvas cv = new Canvas();
cv.setBackground(Color.lightGray);
cv.resize(70, 30);
this.add(cv);
```

Se for necessário redirecionar o teclado para um `Canvas`, é preciso requisitar o foco do mesmo para este componente usando o seu método

`requestFocus()`. Para desenhar, deve-se usar pode-se cadastrar o `Canvas` como fonte de eventos de mouse (`MouseEvent`)

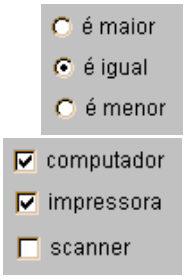
Label

Rótulo de texto flutuante somente-leitura. Usado para rotular outros objetos de um recipiente. Possui métodos para fazer o alinhamento com outros objetos. Exemplo:

```
this.add(new Label("List of Widgets"));
```

Você pode fazer um `Label` responder a eventos da mesma forma que com `Canvas` tratando eventos do teclado com `KeyEvent` e usando `requestFocus()`, e usando eventos do mouse para realizar outras operações.

CheckBox



Caixa de seleção. Possui um método `boolean` que indica se está selecionada ou não (ligada ou desligada). Opcionalmente, pode fazer parte de um `CheckboxGroup`. Apenas um dos objetos `Checkbox` de um `CheckboxGroup` pode ser selecionado ao mesmo tempo.

As três caixas de seleção a seguir, pertencem a o grupo `g` e são exibidas como caixas de rádio. Somente uma das três pode ser selecionada. A segunda está inicialmente selecionada.

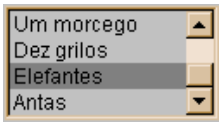
```
CheckboxGroup g = new CheckboxGroup();
this.add(new Checkbox("é maior", g, false));
this.add(new Checkbox("é igual", g, true));
this.add(new Checkbox("é menor", g, false));
```

As caixas de seleção abaixo não pertencem a um grupo. Pode-se selecionar mais de uma.

```
this.add(new Checkbox("computador"));
this.add(new Checkbox("impressora"));
this.add(new Checkbox("scanner"));
```

O evento apenas informa se o estado foi mudado ou não (não diz qual é o estado). O tipo de evento é `ItemEvent`. O manuseador pode ser implementado com `ItemListener` e cadastrado com `addItemListener`.

List



Lista de opções rolante, da qual o usuário pode selecionar uma ou mais opções. Exemplo:

```
List li = new List(4, false);
// false indica que lista não aceita múltiplas seleções
// 4 é o número de opções que será mostrada

li.reshape(0, 0, 100, 100);   // redimensiona
li.add("Um sapo");
li.add("Uma barata");
li.add("Dois ratos");
li.add("Um caipira");
li.add("Um morcego");
li.add("Dez grilos");
li.add("Elefantes");
li.add("Antas");
li.select(3);    // item previamente selecionado é o 4°

this.add(li);
```

Para receber a opção do usuário, implementa-se um `ItemListener`, como em `Checkbox`.

Choice

Lista de opções tipo menu "drop-down". Quando você move o mouse sobre a opção exibida, todas as outras aparecem e você poderá selecionar uma outra opção. Exemplo:

```
Choice opt = new Choice();

opt.add("Daschhund");
opt.add("Dobermann");
opt.add("Collie");
opt.add("Poodle");
opt.add("Vira-Lata");

this.add(opt);
```

Para capturar o evento e obter a seleção do usuário usa-se o mesmo formato utilizado para listas ou `ActionEvent` (para obter a seleção atual).

TextField

Uma linha de texto editável. A maioria dos seus métodos são definidos pela superclasse `TextComponent`, que também são herdados por

`TextArea`, descrito a seguir. Para adicionar um `TextField` no contexto atual, pode-se fazer:

```
TextField tf1 = new TextField("Escreva!"));
TextField tf2 = new TextField(20);
```

`TextField` causa um evento de texto (`TextEvent`) quando o usuário aperta a tecla **ENTER** (ou **RETURN**). Para obter o texto digitado implemente a interface `TextListener` e cadastre o objeto com `addTextListener`. Os métodos de `TextComponent getText()` e `setText(String texto)` servem para obter ou alterar o texto contido no campo de texto.

TextArea

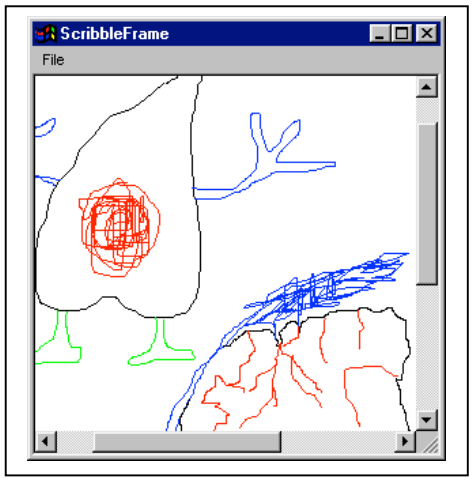
Área de texto editável com várias linhas. É definida informando o número de linhas e colunas e/ou um texto inicial. O exemplo a seguir cria um objeto `TextArea` com 5 linhas visíveis e 20 caracteres de largura:

```
TextArea ta1 = new TextArea(4, 20);
ta1.setText("Você pode digitar aqui!");
```

ou

```
TextArea ta2 = new TextArea("Você pode digitar aqui!",
                                                4, 20);
```

O texto digitado em `TextArea` não gera evento algum, mas fica a disposição para ser lido quando outro evento ocorrer. Por exemplo, pode-se criar um botão que quando apertado, causa um evento e faz com que o conteúdo de `TextArea` seja lido na rotina de manuseio desse evento. Os métodos para ler e escrever texto são os mesmos de `TextComponent` e `TextField`.



ScrollPane

O recipiente `ScrollPane` pode acomodar um único componente filho que pode ser maior que ele mesmo. Na tela aparece uma janela de tamanho fixo que permite a visualização de uma área do componente. Barras de rolamento verticais e/ou horizontais aparecem para permitir a visualização de toda a área.

Para criar um `ScrollPane`, basta instanciá-lo e adicionar um objeto a ele. `ScrollFrame` só suporta um componente e não pode ter um gerente de layouts especificado. Exemplo de uso:

```
ScrollPane pane = new ScrollPane();
pane.setSize(300, 300);
add(pane, "Center");
Panel panel = new Panel ()
panel.resize(500, 500); // Panel é maior que ScrollPane
pane.add(panel);
```

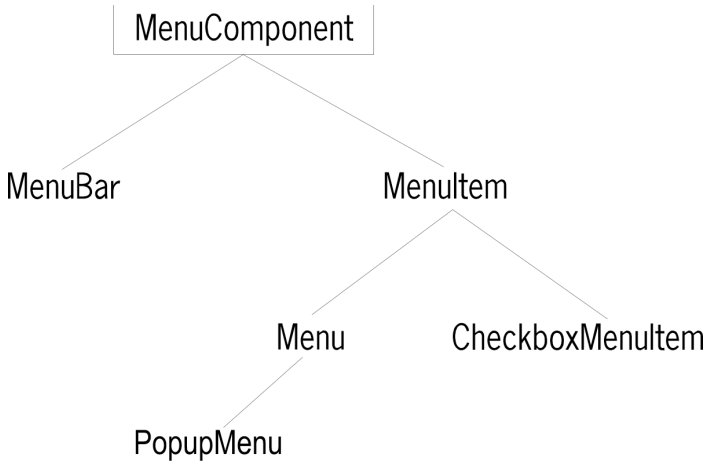
Exercício

1. Desenvolva uma aplicação em Java que consista de um `Frame` e pelo menos um de cada um dos objetos acima. Use um `Panel p` dentro do frame e adicione cada componente no painel usando `p.add(componente)`. Para adicionar o `Panel`, use no construtor:

```
Panel p;
this.add (BorderLayout.CENTER, p);
```

Menus

Menus em Java não são descendentes da classe `Component` mas da uma classe específica, chamada de `MenuComponent`. Através da interface `MenuContainer`, podem ser incluídos apenas em `Frames` (a única exceção é `PopupMenu`). A única forma de incluir menus comuns em uma applet, é através da criação de um objeto `Frame`, separado da applet. A figura abaixo ilustra a hierarquia dos objetos de menu. O `PopupMenu` é o único, da família de menus que pode ser incluído em applets.



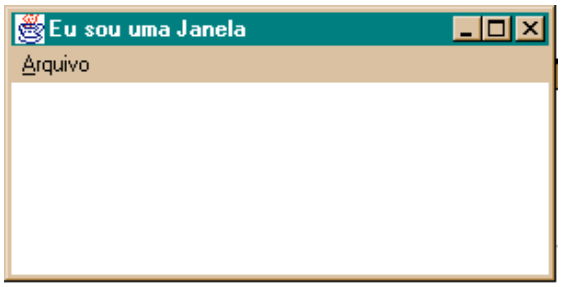
MenuComponent

`MenuComponent`, assim como `Component`, é uma classe raiz da AWT (é uma especialização de `Object`). A classe `MenuComponent` é a superclasse de todas as classes relacionadas com menus. Ela raramente é usada diretamente e seus métodos são utilizados através de suas subclasses `MenuBar` e `MenuItem`.

MenuBar

Cria uma barra de menu que pode ser incluída em um objeto `Frame` (através do método `setMenuBar()`). Menus podem ser adicionados usando o seu método `add()` e `setHelpMenu()`. O exemplo a seguir mostra uma barra de menu dentro de um objeto Frame:

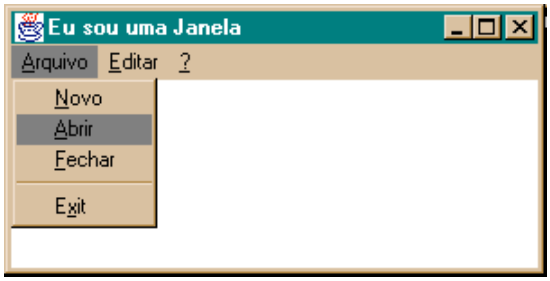
```
import java.awt.*;
public class Quadro {
    static Frame f = new Frame ("Eu sou uma Janela");

    public static void main(String a[]) {
      f.setBounds(300,150,300,200);
      MenuBar mb = new MenuBar(); // cria barra
      mb.add(new Menu("Arquivo")); // adiciona menu
      f.setMenuBar(mb);       // coloca em frame
      f.setVisible(true);
    }
}
```

Menu

Cria um painel tipo "pull-down" que aparece no `MenuBar` (como o menu Arquivo no exemplo acima). Cada `Menu` pode conter várias opções que são objetos do tipo `MenuItem`. No exemplo a seguir, acrescentamos mais dois menus e algumas opções para o primeiro deles:

```
(...)
MenuBar mb = new MenuBar();
  Menu m1 = new Menu("Arquivo"); // Novo menu 1
    m1.add(new MenuItem("Novo"));  // Criação e
    m1.add(new MenuItem("Abrir")); // adição das
    m1.add(new MenuItem("Fechar"));// opções do
    m1.addSeparator();              // menu 1
    m1.add(new MenuItem("Exit"));
mb.add(m1);
  Menu m2 = new Menu("Editar"); // Novo menu 2
mb.add(m2);
  Menu m3 = new Menu("?");      // Novo menu 3
mb.setHelpMenu(m3);
(...)
```

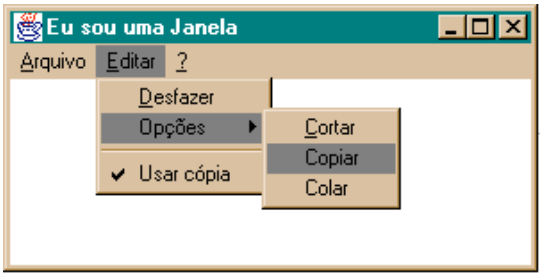


MenuItem

Esta classe define um item de menu com um determinado rótulo textual. O método de instância `add()` da classe `Menu` é utilizado para adicioná-lo a um menu. Como `Menu` também é um `MenuItem`, é possível incluí-lo dentro de outros

menus. `CheckBoxMenuItem` é uma especialização de `MenuItem`, que cria um item que pode ser ligado ou desligado. O exemplo a seguir ilustra essas opções:

```
(...)
  Menu m2 = new Menu("&Editar"); // Novo menu m2
    m2.add(new MenuItem("Desfazer"));
    Menu m4 = new Menu("Opções"); // Novo menu m4
      m4.add(new MenuItem("Cortar"));
      m4.add(new MenuItem("Copiar"));
      m4.add(new MenuItem("Colar"));
    m2.add(m4);     // m4 é sub-menu (item) de m2
    m2.addSeparator();
    m2.add(new CheckboxMenuItem("Usar cópia"));
mb.add(m2);
(...)
```



Para capturar e lidar com um evento produzido pela seleção de uma opção de menu deve-se usar a estrutura para capturar `ActionEvents`.

Menus PopUp

Em Java 1.0, menus só podiam ser adicionados em Frames. No JDK 1.1 se permite que menus pulem de qualquer parte de um componente. São os "pop-up menus". Para implementá-los basta instanciar a classe `PopupMenu`, acrescentar itens `MenuItem` ou submenus com a classe Menu e adicioná-los ao componente usando o seu método `add()`.

```
PopupMenu pop = new PopupMenu();     // cria menu
pop.add(new MenuItem("Recortar");          // adiciona itens
pop.add(new MenuItem("Colar");
pop.add(new MenuItem("Copiar");
pop.addSeparator();
pop.add(new Menu("Mover para"));
add(pop);  // adiciona ao componente
```

Para fazer o menu aparecer, é preciso capturar o evento que dispara o menu popup ("pop-up trigger") E, nele, chamar o método `show()` de `PopupMenu` nas coordenadas onde o mouse foi acionado:


```
public void processMouseEvent(MouseEvent e) {
if (e.isPopupTrigger()) {
    pop.show(this, e.getX(), e.getY());
 super.processMouseEvent(e);
}
```

A forma de implementação do `PopupTrigger` varia entre plataformas (no Windows, ele é disparado ao se clicar o botão direito do mouse).

Exercícios

2. Incremente a aplicação que você fez no primeiro exercício com menus e pop-up menus.

1.4. Layouts

As classes descritas a seguir implementam a interface `LayoutManager` ou `LayoutManager2`. Estas interfaces definem os métodos necessários para que uma classe possa arrumar objetos do tipo `Component` dentro de um objeto do tipo `Container`.

Layouts em Java são uma forma independente de plataforma de posicionar objetos em uma GUI (você pode usá-los se quiser, ou desligá-los, mas fazendo isto, você terá que posicionar os componentes definindo os pixels que compõem suas coordenadas).

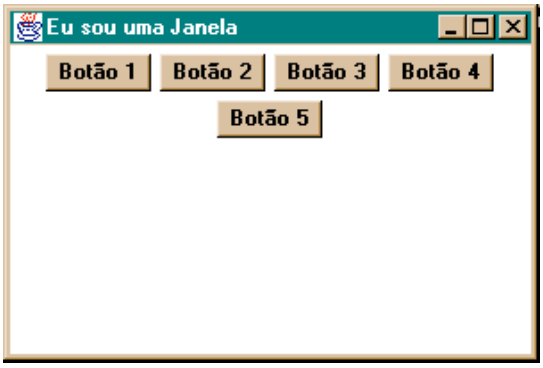
FlowLayout

Organiza os objetos da esquerda para a direita em linhas. Quando acaba uma linha, segue para a próxima. É usado por *default* na classe `Panel` (e pela sua subclasse `Applet`). Na classe `Frame` pode ser definido usando `setLayout(new FlowLayout())`. Exemplo:

```
public class Quadro extends Frame{

    public Quadro() {
        super("Eu sou uma Janela");
        setSize(300,200);

        setLayout(new FlowLayout());
        add(new Button("Botão 1"));
        add(new Button("Botão 2"));
        add(new Button("Botão 3"));
        add(new Button("Botão 4"));
```



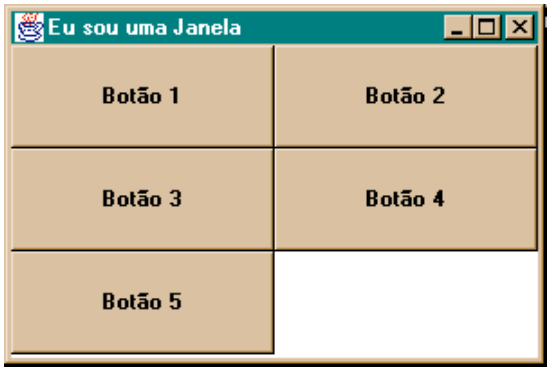
```
        add(new Button("Botão 5"));

        setVisible(true);
    }

    public static void main(String a[]) {
        new Quadro();
    }
}
```



GridLayout

Divide o recipiente em um número especificado de linhas e colunas e arruma os componentes neles da esquerda para a direita e de cima para baixo. Exemplo:

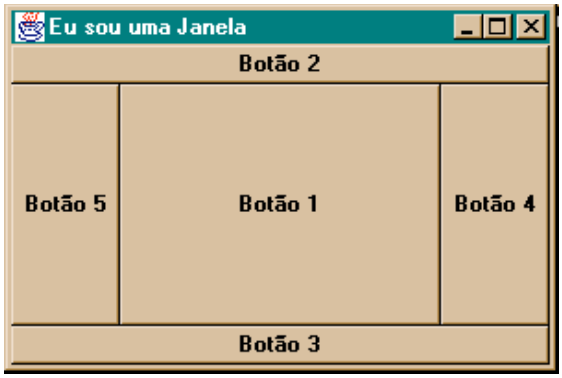
```
        setLayout(new GridLayout(3, 2));
        add(new Button("Botão 1"));
        add(new Button("Botão 2"));
        add(new Button("Botão 3"));
        add(new Button("Botão 4"));
        add(new Button("Botão 5"));

        setVisible(true);
```



BorderLayout

Arruma os componentes nas laterais do recipiente, usando os nomes `North`, `South`, `East`, `West` e `Center` ou as constantes `BorderLayout.NORTH`, `SOUTH`, `EAST`, `WEST` e `CENTER` no método `add()`, para informar a posição de cada objeto. Exemplo:

```
setLayout(new BorderLayout());
add("Center", new Button("Botão 1"));
add("North", new Button("Botão 2"));
add("South", new Button("Botão 3"));
add("East", new Button("Botão 4"));
add("West", new Button("Botão 5"));

setVisible(true);
```



CardLayout

Só é possível visualizar na tela um componente de cada vez, se eles forem organizados por `CardLayout`. O uso mais comum, é definir objetos `Panel` (possivelmente com *layouts* diferentes), cada qual com seus componentes e organizá-los com `CardLayout`. Pode-se, então, determinar botões específicos para mostrar cada um deles, um por vez. A applet a seguir mostra, dependendo do botão selecionado, um dos dois painéis mostrados na figura:

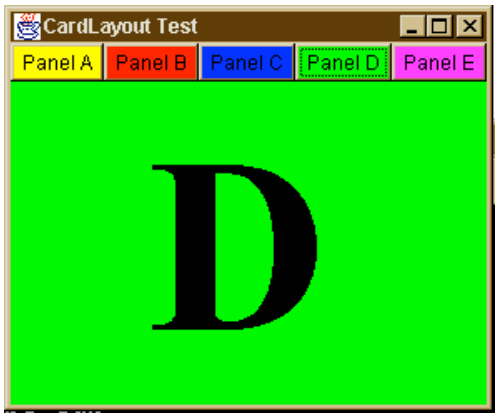
```
import java.awt.*;
import java.awt.event.*;

public class CardLayoutTest extends Frame {

   private Button b1, b2, b3, b4, b5;
   private CardLayout tabs;

   public CardLayoutTest() {

      super("CardLayout Test");
```



```
// Button's panel
Panel topPanel = new Panel(new FlowLayout(0,0,FlowLayout.LEFT));

b1 = new Button("Panel A");
b1.setBackground(Color.yellow);
b2 = new Button("Panel B");
b2.setBackground(Color.red);
b3 = new Button("Panel C");
b3.setBackground(Color.blue);
b4 = new Button("Panel D");
b4.setBackground(Color.green);
b5 = new Button("Panel E");
b5.setBackground(Color.magenta);

topPanel.add(b1);
topPanel.add(b2);
topPanel.add(b3);
topPanel.add(b4);
topPanel.add(b5);

add(BorderLayout.NORTH, topPanel);

// Card Panel
tabs = new CardLayout();
final Panel lowerPanel = new Panel(tabs);
lowerPanel.setFont(new Font("Helvetica", Font.BOLD, 150));

Panel p1 = new Panel();
p1.setBackground(Color.yellow);
p1.add(new Label("A"));

Panel p2 = new Panel();
p2.setBackground(Color.red);
p2.add(new Label("B"));

Panel p3 = new Panel();
p3.setBackground(Color.blue);
p3.add(new Label("C"));

Panel p4 = new Panel();
p4.setBackground(Color.green);
p4.add(new Label("D"));

Panel p5 = new Panel();
p5.setBackground(Color.magenta);
p5.add(new Label("E"));

lowerPanel.add("pan1", p1);
lowerPanel.add("pan2", p2);
lowerPanel.add("pan3", p3);
lowerPanel.add("pan4", p4);
lowerPanel.add("pan5", p5);
```

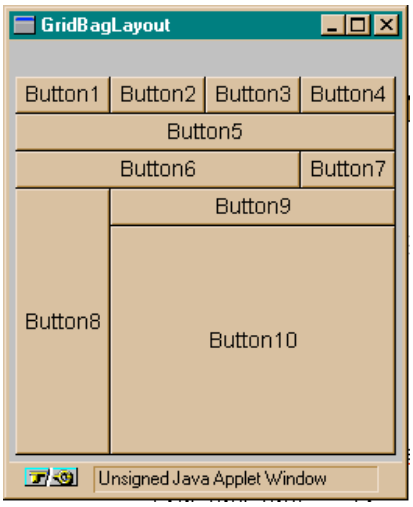


```
        add(BorderLayout.CENTER, lowerPanel);
        tabs.show(lowerPanel,"pan1");

        // eventos
        b1.addActionListener (new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              tabs.show(lowerPanel,"pan1");
           }
        });

        b2.addActionListener (new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              tabs.show(lowerPanel,"pan2");
           }
        });

        b3.addActionListener (new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              tabs.show(lowerPanel,"pan3");
           }
        });

        b4.addActionListener (new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              tabs.show(lowerPanel,"pan4");
           }
        });

        b5.addActionListener (new ActionListener() {
           public void actionPerformed(ActionEvent e) {
              tabs.show(lowerPanel,"pan5");
           }
        });

        // Frame
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new CardLayoutTest();
    }
}
```

GridBagLayout

É o mais complicado dos gerentes de layout. Divide o recipiente em uma grade de linhas e colunas, com dimensões variáveis definidas por um objeto `GridBagConstraints`. Com este gerente de layout, é possível organizar os componentes de uma maneira elegante e garantir que eles não se sobreponham. O `GridBagLayout` não será abordado neste curso.



Outros layouts

Usando os layouts disponíveis para organizar componentes em `Panels` e depois utilizar esses mesmos `Panels` em outros `Panels`, com outros layouts, é possível colocar componentes AWT onde você quiser. Você também tem a opção de não utilizar layout algum. Para isto é somente definir o layout como nulo:

```
setLayout(null);
```


1.5. Você terá então que posicionar cada objeto no recipiente, de acordo com as suas coordenadas em pixels. Terá também que definir os tamanhos e espaços entre os componentes. Quando se usa um layout, não existe liberdade. Eles são tirânicos. Não adianta usar `setSize()` em um `BorderLayout`, assim como não adianta usar um `setLocation()` em `FlowLayout`. Simplesmente nada acontece, pois as regras do algoritmo de posicionamento e redimensionamento do layout sempre têm prioridade. A opção de desligar os layouts, embora ofereça liberdade absoluta, diminui a portabilidade do seu programa, pois não é possível prever como a janela irá se apresentar quando exibida em outro sistema operacional.

JAVA 6

Entrada/saída e rede

ESTE MÓDULO MOSTRA AS APIS JAVA PARA ACESSO A DISCO E À REDE. Uma visão geral do modelo de multithreading da linguagem também é oferecida.

Tópicos abordados neste módulo

- Threads (overview)
- Entrada e saída
- Fluxos de dados (streams)
- Descritores de arquivos
- Métodos para leitura e escrita de dados
- Comunicações em rede
- Socket, DatagramSocket e outras de componentes TCP IP
- Servidor multithreaded

Índice

Objetivos

- No final deste módulo você deverá ser capaz de:
- Saber como funciona o modelo de threads da linguagem Java e saber usar threads para criar um processo paralelo.
- Abrir um arquivo para leitura ou escrita e transferir caracteres, bytes, strings ou objetos.
- Converter um objeto de forma que ele possa ser salvo ou enviado pela rede
- Saber como ter acesso aleatório a um arquivo
- Saber usar um fluxo de dados e conhecer as principais classes de entrada e saída em Java
- Abrir (utilizar) um soquete de serviços TCP IP
- Instalar um soquete de servicos TCP IP
- Identificar as classes principais do pacote java.net

6.1. Programação Concorrente (multithreading)

JAVA POSSUI SUPORTE NATIVO para *multithreading*, isto é, a possibilidade de ter múltiplas linhas simultâneas de execução em um mesmo programa. Com este recurso, é possível controlar cada linha de execução, suas prioridades, quando vão ocorrer, estabelecer travas, etc. dentro do programa.

Um *thread* pode ser definido como uma CPU virtual que pode operar em dados próprios ou que são compartilhados com outras linhas de execução. É fácil criar um *thread* em Java. O primeiro passo é definir o código que ele irá executar.

Isto é feito fazendo com que a classe que define o thread implemente a interface `Runnable`, que tem um único método a implementar: `public void run()`:

```
public class Paralelo implements Runnable {
    int x;
    public void run() {
        while (true) {
            System.out.println(++x + " carneirinhos...");
        }
    }
}
```

Para criar um thread, agora, basta fazer:

```
Runnable r = new Paralelo();
Thread contaCarneiros = new Thread(r);
```

Depois de criado, o objeto pode ser configurado e executado:

```
contaCarneiros.setPriority(Thread.MAX_PRIORITY);
contaCarneiros.start();
```

O thread acima opera em cima de dados locais (x), que não são compartilhados por outros threads. Ele poderia também operar em dados de instância, que outros threads poderiam alterar. Para evitar problemas de corrupção de dados, nesses casos, é preciso usar blocos sincronizados, que veremos adiante.

O objeto `Runnable` resolve todos os nossos problemas, mas é muito mais fácil usar threads estendendo a classe `Thread`. Assim teremos acesso direto a vários métodos de controle do thread que seriam mais difíceis de usar com `Runnable`. `Runnable` é indicado quando é impossível estender `Thread`, quando a classe já estende outra classe. O programa acima pode ser rescrito estendendo a classe `Thread`, da forma:

```
public class Paralelo extends Thread {
    int x;
    public void run() {
        while (true) {
            System.out.println(++x + " carneirinhos...");
        }
    }
}
```

Para criar o objeto, pode-se fazer:


```
Paralelo p = new Paralelo();
p.start();
```

Uma vez criado um objeto `Thread`, ele pode ser manipulado através de diversos métodos e funções estáticas. Os principais métodos de instância utilizáveis por objetos da classe `Thread` são:

- **`start()`:** inicia uma nova linha de controle com base nos dados do objeto `Thread` e invoca o seu método `run();`
- **`run()`:** contém toda a implementação do `Thread`. Deve ser redefinido em uma sub-classe de `Thread` ou em qualquer classe que implemente `Runnable` (`run()` é quase vazio na classe `Thread` para classes que a sobrepõem. Não faz nada)
- **`stop()`:** interrompe a execução da linha de controle e mata o thread. Depois de morto um thread não pode mais ressuscitar.
- **`suspend()`:** suspende temporariamente a linha de controle.
- **`resume()`:** reinicia uma linha de controle que foi suspensa.

As principais funções (métodos estáticos) disponíveis na classe `Thread` são:

- **`sleep(`*`milisegundos`*`)`:** suspende a execução do thread ativo neste momento por um determinado período de no mínimo `milisegundos`.
- **`yield()`:** dá a preferência a outras linhas que estiverem esperando pela oportunidade de executar.

Estes métodos e funções alteram os estados de um thread, que são quatro. Um thread vivo pode assumir três estados: *executando*, *pronto* (ready) e *esperando* (suspenso, dormindo, aguardando notificação ou bloqueado). O quarto estado é *morto*, que ocorre quando um thread termina ou é assassinado com uma invocação do método `stop()`. Um thread está vivo enquanto está executando o seu método `run()`. Quando um thread morre, ele não volta à vida mas todos os seus outros métodos podem ainda ser chamados.

Antes de discutir esses estados e como fazer para controlá-los, vejamos alguns exemplos com threads. O exemplo a seguir, do livro "The Java Programming Language", de James Gosling e Ken Arnold, ilustra a extensão da classe `Thread` e a posterior criação de dois objetos que executam simultaneamente:

```
// Exemplo do livro "The Java Programming Language" (p.161)
```



```
class PingPong extends Thread {
    String palavra;
    int atraso;

    PingPong (String oQueDizer, int tempoAtraso) {
        palavra = oQueDizer;
        atraso = tempoAtraso;
    }

    public void run() {
        try {
            for (;;) {
                System.out.print(palavra + " ");
                sleep(atraso);  // espera até a próxima;
            }
        } catch (InterruptedException e) {
            return;     // fim deste thread
        }
    }

    public static void main(String[] args) {
        new PingPong("ping", 33).start();   // 1/30 s
        new PingPong("PONG", 100).start();  // 1/30 s
    }
}
```

Compile e rode o exemplo. Depois experimente acrescentar novas linhas de execução (criando novos objetos `PingPong`), com tempos diferentes.

Nem sempre é possível estender a classe `Thread`. Toda applet, por exemplo, tem que estender a classe `Applet` e como Java não permite herança múltipla, uma classe não pode estender ao mesmo tempo a classe `Applet` e `Thread`.

Uma saída é usar uma classe para definir a `Thread` e outra para usar objetos dela. Pode-se ainda fazer tudo na mesma classe através da criação de um objeto `Runnable`, passado ao construtor do `Thread` como no exemplo a seguir.

```
// Exemplo do livro "The Java Programming Language" (p.177)

class RunPingPong implements Runnable {
    String palavra;
    int atraso;

    RunPingPong (String oQueDizer, int tempoAtraso) {
        palavra = oQueDizer;
```



```
        atraso = tempoAtraso;
    }

    public void run() {
        try {
           for (;;) {
             System.out.print(palavra + " ");
             Thread.sleep(atraso);  // espera até próxima;
           }
        } catch (InterruptedException e) {
            return;     // fim deste thread
        }
    }

    public static void main(String[] args) {
        Runnable ping = new RunPingPong("ping", 33);
        Runnable pong = new RunPingPong("PONG", 100);
        new Thread(ping).start();
        new Thread(pong).start();
    }
}
```

Este exemplo faz o mesmo que o anterior só que não estende a classe `Thread`. Desta forma pode ainda estender uma classe (`Applet`, por exemplo). Como implementa a interface `Runnable` (que só declara o método `run()`), só precisa redefinir `run()` e depois passar os objetos `Runnable` como argumentos na construção de objetos `Thread`. Desta forma, cria-se um objeto `Thread` com o método `run()` definido fora dele.

Ciclos de vida

Threads têm um ciclo de vida que inicia quando são criados e termina quando morrem. Durante a sua vida, um *thread* pode viver em três estados:

- Executando, quando ele consegue um espaço da CPU para realizar o que foi programado para fazer
- Esperando, quando está inativo por alguma razão.
- Pronto, quando não espera nada a não ser a oportunidade de executar pela CPU.

Devido ao compartilhamento de tempo em uma CPU, um *thread* está sempre alternando com os outros *threads* entre o estado pronto e o estado executando. Vários *threads* podem estar ao mesmo tempo no estado pronto,

esperando que a CPU lhe dê a vez. Se houver um *thread* de maior prioridade na sua frente, ele terá que esperar. Se não houver prioridade, ele não pode ter certeza quando vai ser chamado. Não há garantia que o *thread* que espera há mais tempo será chamado.

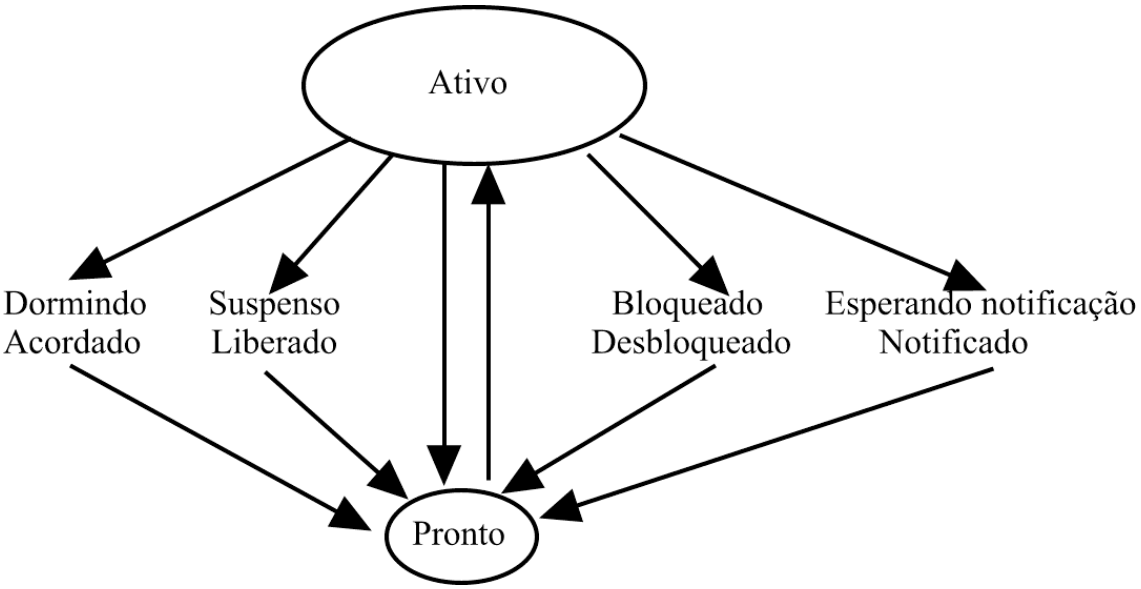
O método `setPriority()` pode ser chamado para definir a prioridade de threads. Os valores passados podem ser qualquer valor de 1 a 10.

É importante observar que o funcionamento de prioridades em *threads* é dependente de plataforma. Algoritmos que dependem das prioridades dos *threads* poderão gerar resultados imprevisíveis.

Controle de threads

Há várias maneiras de controlar a mudança de estado de um *thread*. As formas são:

- Dando a preferência (chamando a função `yield()`)
- Dormindo e acordando (chamando a função `sleep(tempo)`)
- Suspendendo (`suspend()`) e depois reiniciando (`resume()`)
- Bloqueando (efeitos externos diversos e interrupções)
- Esperando (`wait()`) e sendo notificado (`notify()`)



A figura abaixo ilustra os estados de um *thread* vivo.

`Yield()` é uma função estática. Opera no *thread* ativo. É usado pelo *thread* que ocupa a CPU quando quer dar uma oportunidade aos outros *threads*. `yield()` faz com que ele deixe a CPU e vá para o estado pronto. se houver algum *thread* interessado na CPU, este ganha a vez. Se não, ele volta para a CPU.

`Sleep()` coloca um *thread* para dormir durante um determinado tempo. É um método estático e sempre opera no objeto atual. Depois que um thread acorda, ele não volta a ocupar a CPU imediatamente. Ele passa para o estado pronto e aguarda a sua vez.

Com `suspend()`, um *thread* entra em um estado de suspensão de suas atividades e fica lá até que outro *thread* invoque o método `resume()` sobre ele. Quando isto acontece, ele também não volta imediatamente à CPU. Fica no estado pronto até que possa voltar à ativa.

O bloqueio ocorre quando um fator externo ou uma interrupção (usando `interrupt()`) causa a espera do *thread*. Quando a causa do bloqueio se vai, novamente o *thread* entra no estado de prontidão e espera sua vez pela CPU.

Sincronização

Java usa a palavra-chave `synchronized` para declarar blocos de código, métodos e classes que devem ter os dados em que operam travados, para evitar acessos simultâneos e a conseqüente corrupção de dados.

Se um thread invoca um método "`synchronized`" sobre um objeto, esse objeto é *travado*. A sincronização garante que a execução de duas linhas serão mutualmente exclusivas no tempo. Os métodos sincronizados esperam enquanto um outro termina seu trabalho antes de operar no mesmo conjunto de dados. Os métodos `wait()`, `notify()` e `notifyAll()` só podem ocorrer dentro de blocos `synchronized`!

A sincronização resolve um problema mas pode causar outro. Sempre que houver duas linhas e dois objetos com travas, poderá acontecer um *deadlock*, onde cada objeto espera pela liberação da trava do outro.

Por exemplo: Se o objeto A tem um método sincronizado que invoca um método sincronizado no objeto B, que por sua vez tem um outro método sincronizado invocando um método sincronizado no objeto A, ambos irão ficar esperando eternamente pelo outro.

Java não detecta nem evita *deadlocks*. O programador deve criar mecanismos no programa que os evite.

Uma forma de evitar *deadlocks* e controlar melhor programas que usam muitas linhas de execução paralelas é agrupá-las em objetos `ThreadGroup`. Esta classe permite o agrupamento de múltiplas linhas de controle em um único objeto e então controlar vários *threads* ao mesmo tempo, como um grupo, estabelecer limites e garantir maior segurança.

Exercícios

1. Altere os tempos de execução dos threads nos programas-exemplo deste capítulo.
2. Crie novos threads com textos diferentes e tempos diferentes.
3. Crie uma pilha com dois métodos `pop()` e `push()`. Eles devem respectivamente extrair e colocar um caractere na pilha.
4. Crie uma classe `Produtor` que estende `Thread` que produz caracteres e os armazena em uma pilha.
5. Crie uma classe `Consumidor` que estende `Thread` que extrai caracteres de uma pilha.
6. Crie uma terceira classe para simular o ato de produzir e consumir.
7. Torne os métodos `push()` e `pop()` `synchronized` para evitar a corrupção de dados. Implemente um mecanismo que use `wait()` e `notify()`.

6.2. Persistência e Entrada/Saída

A troca de informações entre aplicações remotas ou partes de uma aplicação consiste em ter um dos lados, em algum momento, fazendo papel de servidor, enquanto o outro utiliza seus serviços fazendo papel de cliente. Estas operações dependem do suporte da biblioteca `java.io`. Neste capítulo, apresentamos uma breve introdução ao pacote `java.io`, o funcionamento dos mecanismos de entrada e saída de Java e a serialização de objetos. No final, apresentaremos um exemplo que utiliza todos estes recursos na construção de uma aplicação simples de banco de dados armazenado em arquivo com acesso aleatório.

Fluxos de dados (data streams)

Para compreender os mecanismos usados na comunicação entre objetos em rede, é essencial conhecer o funcionamento dos fluxos de entrada e saída, suportados pelas classes do pacote `java.io`. Esses objetos permitem a transferência de dados (bytes) e caracteres Unicode a partir de fontes como arquivos, soquetes de rede, etc. Implementam métodos e filtros usados para recuperar e transmitir a informação da forma mais eficiente.

As classes que suportam fluxos de dados em Java têm todas o sufixo '`Stream`'. São de dois tipos: fluxos de entrada: `InputStream`; e de saída: `OutputStream`. Com elas pode-se construir objetos que têm a tarefa de receber ou produzir fluxos de *bytes*. Os objetos de fluxos de dados são unidirecionais e devem ser "conectados" a uma fonte ou destino de onde ou para onde devem enviar ou receber os bytes.

Pode-se comparar um fluxo de bytes a um fluxo de água que conduz o líquido a partir de uma caixa d'água (fonte persistente). Pode-se obter um fluxo de água a partir de uma caixa d'água conectando-se um cano à uma de suas saídas. De forma análoga, a para ler os *bytes* de um arquivo deve-se conectar um `FileInputStream` a ele. Veja a figura 6-1.

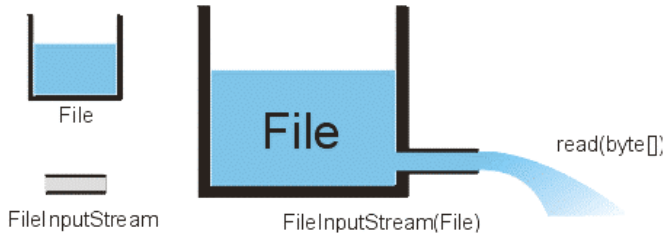


Figura 6-1

A forma de implementar o exemplo acima em Java é:

```
File tanque = new File("agua.txt");   // objeto do tipo File
FileInputStream cano =                // referência FileInputStream
       new FileInputStream(tanque);   // cano conectado no tanque
byte octeto = cano.read();            // lê um byte a partir do cano
```

Depois de abertos os fluxos de dados, eles podem ser usados (através da invocação de métodos) e fechados, invocando o seu método `close()`, presente, juntamente com os métodos `read()` e `write()`, nas classes `InputStream` e `OutputStream` e herdado por todas as subclasses do tipo `Stream`. Além de `read()` e `write()`, que lêem ou gravam um byte de cada vez os fluxos de dados `InputStream` e `OutputStream` possuem métodos que permitem ler ou gravar

blocos maiores de bytes de cada vez, tornando a transferência mais eficiente. O objeto `FileInputStream` fornece um método para ler bytes individuais ou conjuntos de bytes de um arquivo. Geralmente, é preciso fazer uma transformação adicional. Por exemplo, se os dados estiverem no formato texto, será preciso reconhecer os caracteres (que ocupam pelo menos dois bytes), as linhas, parágrafos, etc. Para facilitar o trabalho de transformação, o pacote `java.io` conta com vários *filtros* que transformam bytes em uma variedade de formatos úteis. Os filtros são subclasses de `java.io.FilterInputStream` e `FilterOutputStream`. Recebem um fluxo de dados na entrada (na construção do objeto) e devolvem o fluxo transformado na saída (na invocação de métodos de leitura/gravação). Com isto é possível tratar os dados em um nível mais alto, escondendo a complexidade por trás do tratamento de informações.

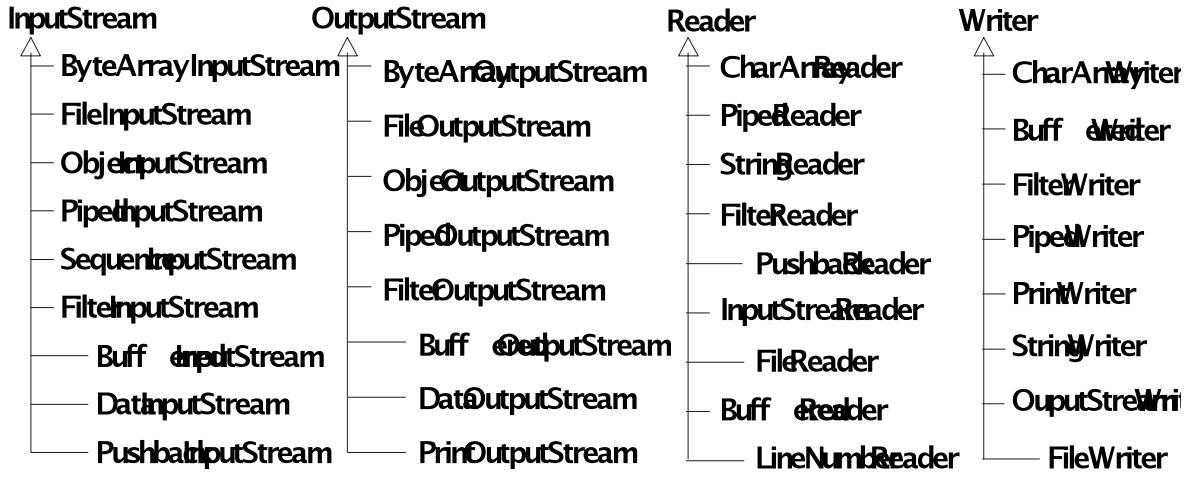


Figura 6-2 – Hierarquia de fluxos de dados e caracteres na plataforma Java 2

As classes com sufixo `Reader` ou `Writer` são usadas, respectivamente, para ler e gravar *caracteres* Unicode (16 bits), em vez de *bytes* (8 bits). Como caracteres Java possuem pelo menos dois bytes cada, a leitura byte-a-byte pode não fornecer toda a informação necessária para identificar um caractere. Portanto, é preciso converter conjuntos de bytes em caracteres de acordo com o formato de codificação usado. Uma forma prática é utilizar as classes do tipo `Reader` ou `Writer`.

Para converter fluxos de *bytes* em fluxos de *caracteres* (dois bytes cada) durante a leitura do arquivo, podemos ler a partir de um filtro do tipo

`InputStreamReader`, "conectado", a um objeto `FileInputStream`. Os métodos de `InputStreamReader` lidam diretamente com caracteres Unicode. Veja como poderíamos implementar isto em Java:

```
File tanque = new File("agua.txt");   // objeto do tipo File
FileInputStream cano =                // referência FileInputStream
       new FileInputStream(tanque);   // cano conectado no tanque
InputStreamReader chf =
       new InputStreamReader(cano);   // filtro chf conectado no cano
char letra = chf.read();         // lê um char a partir do filtro chf
```

Os fluxos de dados podem ser colocados em cascata indefinidamente até se obter um formato adequado à manipulação dos dados. Aplicando mais um filtro à saída de `InputStreamReader`, por exemplo, pode-se lidar com uma linha de cada vez, através do método `readLine()`. Veja o exemplo abaixo e a figura 6-3.

```
File tanque = new File("agua.txt");   // objeto do tipo File
FileInputStream cano =                // referência FileInputStream
       new FileInputStream(tanque);   // cano conectado no tanque
InputStreamReader chf =
       new InputStreamReader(cano);   // filtro chf conectado no cano
BufferedReader br =
       new BufferedReader (chf);      // filtro br conectado no chf
String linha = br.readLine();         // lê linha de texto a de br
```

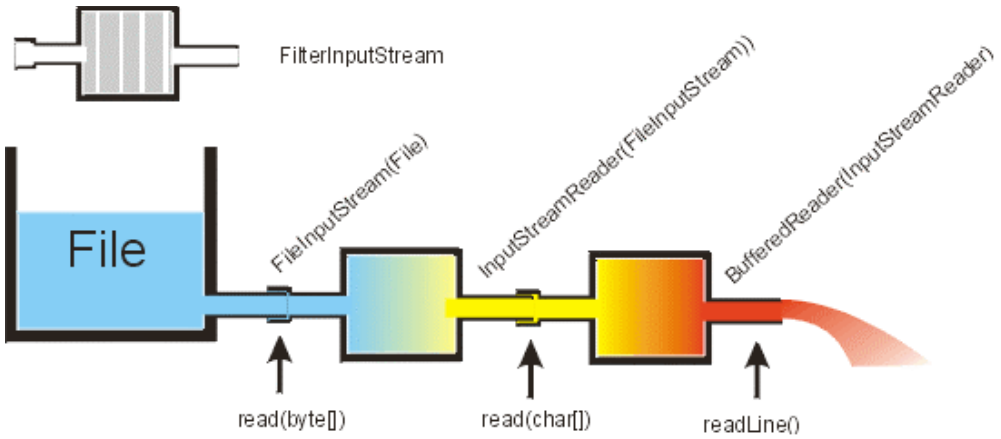


Figura 6-3 – Leitura 'filtrada' a partir de um arquivo

Os filtros de dados são extremamente úteis e praticamente indispensáveis em Java. Para desenvolver objetos de fluxo personalizados, o melhor caminho é criar subclasses a partir das classes de fluxos de dados que mais se aproximam do formato desejado. Com este procedimento pode-se ocultar a complexidade

dessas transferências e oferecer formas mais simples de obter os dados, isolando a formatação de informações do restante da aplicação.

Todas as operações de entrada e saída podem não ocorrer por diversos motivos: arquivo não encontrado, falta de permissão, violação de compartilhamento, falta de memória, etc. Java trata cada um desses problemas em exceções que são subclasses de `java.io.IOException`. A grande maioria das operações de entrada e saída provocam `IOException` ou uma das suas subexceções e o código onde são usadas deve levar isto em conta ora tratando a exceção (com try-catch) ou propagando-a para outros métodos (cláusula `throws`).

Descritores de arquivos

Além dos fluxos, o pacote `java.io` contém ainda duas classes que representam arquivos em disco. `File` e `RandomAccessFile`. A primeira, mencionada nos exemplos apresentados, descreve um arquivo e possui vários métodos para manipular com eles. A segunda classe implementa um ponteiro que pode ler ou escrever aleatoriamente em um arquivo. Nos exemplos apresentados neste capítulo, utilizamos os dois tipos.

A classe File

A forma mais simples de criar um descritor de arquivos é através de seu construtor que recebe um String informando o nome ou caminho do arquivo:

```
File f = new File("c:\tmp\lixo.txt");
```

Os métodos disponíveis na classe `File` permitem a realização de operações com arquivos e diretórios. Arquivos comuns não são criados com métodos da classe `File`. Eles são criados somente quando se grava alguma coisa através de um `FileOutputStream`. Diretórios, porém, podem ser criados diretamente usando o método `mkdir()`:

```
File diretorio = new File("c:\tmp\cesto");
diretorio.mkdir(); // cria diretório se possível
File arquivo = new File("c:\tmp\cesto\lixo.txt");
FileOutputStream out = new FileOutputStream(arquivo);
out.write("lixo inicial");     // se arquivo não existe, tenta criar
```

Tanto a criação de diretórios como a criação de arquivos podem não ocorrer se houver uma `IOException`. O código acima deve estar presente em um

bloco `try-catch` ou em um método que declara a possibilidade de ocorrência de `IOException`.

Os outros métodos de `File` realizam testes, obtém informações e alteram arquivos existentes. A seguir, os principais construtores e métodos. Os nomes dos métodos são bastante auto-explicativos, portanto, não detalharemos seu funcionamento.

Construtores:

```
public File(String caminho) throws NullPointerException;
public File(String caminho, String nome);
public File(String diretório, String caminho);
```

Principais Métodos:

```
public boolean exists();
public boolean delete();
public boolean isDirectory();
public boolean ifFile();
public boolean mkdir();
public boolean mkdirs();
public boolean renameTo(File novo_arquivo);
public boolean equals(Object obj);

public String getAbsolutePath();
public String getName();
public String getPath();
public String[] list();
public String[] lins(FilenameFilter filtro);

public long lastModified();
public long length();
```

O separador usado para descrever caminhos de subdiretórios é um caractere dependente de plataforma. Java oferece, porém, uma constante que contém o separador usado pelo sistema. `File.separator` contém uma `String` que representa o "/", "\" ou outro separador de arquivos dependendo do sistema onde a aplicação está rodando.

RandomAccessFile

A classe `java.io.RandomAccessFile` representa um descritor especial de arquivos que contém um ponteiro que permite acesso aleatório a partes do arquivo representado. Não é possível conectar fluxos de entrada e saída em um

`RandomAccessFile` pois não é um `File` e não funciona com acesso seqüencial proporcionado pelas classes `InputStream` e `OutputStream`. `RandomAccessFile` oferece um meio completamente independente para ler e gravar informação de e para arquivos. Implementa as interfaces `DataInput` e `DataOutput` (as mesmas implementadas por `DataOutputStream` e `DataInputSttream`) que oferece métodos para gravar e ler tipos primitivos diretamente.

A criação de um `RandomAccessFile` requer como argumentos um arquivo ou nome de arquivo e um `String` indicando se o arquivo será aberto apenas para leitura ou tanto para leitura como gravação:

```
RandomAccessFile raf = new RandomAccessFile("lista1.txt", "r");
File arquivo = new File("lista2.txt");
RandomAccessFile raf = new RandomAccessFile(arquivo, "rw");
```

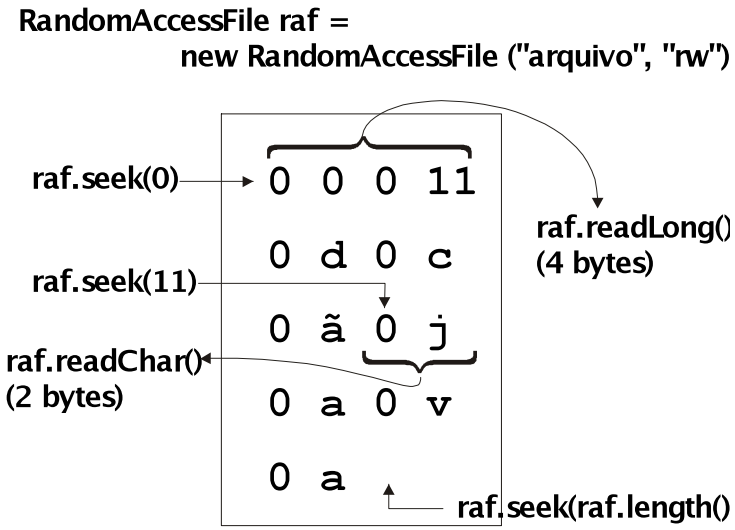


Figura 6-4: RandomAccessFile

O método `seek(posição)` permite o acesso aleatório ao arquivo. Ele posiciona o ponteiro na posição selecionada e a partir daí pode ler os bytes seguintes ou gravar, sobrepondo a informação seguinte. A partir de uma posição, pode-se avançar byte a byte ou em blocos maiores usando os métodos de `DataInput` e `DataOutput` que lêem e gravam tipos primitivos.

O código abaixo (ilustrado pela figura 6-4, ao lado) mostra o uso de `RandomAccessFile` para abrir um arquivo existente (ou criar um arquivo inexistente), ler um número inteiro longo armazenado na primeira linha que indica a posição de um caractere (que deve ser lido e copiado para o fim do arquivo), copiar o caractere e depois incrementar o número da primeira linha para que contenha a posição do próximo caractere a ser copiado:

```
RandomAccessFile raf = new RandomAccessFile("dados.txt", "rw");
raf.seek(0);                    // início do arquivo
long posicao = raf.readLong();   // posição onde encontrar letra
```



```
raf.seek(posicao);            // ponteiro na nova posição
char letra = raf.readChar();  // lê caractere em posicao
long novaPos = raf.getFilePointer();  // posição do próximo char
raf.seek(raf.length());       // fim do arquivo
raf.writeChar(letra);         // copia letra para fim do arquivo
raf.seek(0);                  // volta ao inicio do arquivo
raf.writeLong(novaPos);       // incrementa posição
```

Serialização de objetos

Para salvar um objeto em disco ou em outra forma de armazenamento persistente, é preciso salvar o estado individual dos tipos primitivos que o compõem. Se o objeto contém outros objetos, estes também precisam ser decompostos em tipos primitivos e assim por diante, até que se possa armazenar os dados usando os métodos `write()`, `writeInt()`, `writeChar()`, etc. Se um objeto não tiver um estado transiente (como soquetes ou threads) e não possuir como membros objetos transientes, poderá ser declarado "serializável" e se beneficiar do recurso de "serialização" de objetos oferecida por Java. A serialização de objetos converte automaticamente todo o objeto em uma representação portátil que pode ser enviada através de uma rede, armazenada em disco ou em outro meio. Com a serialização, torna-se trivial a tarefa de criar objetos persistentes.

Para tornar um objeto "serializável", basta declarar, na sua declaração de classe, que a mesma implementa a interface `java.io.Serializable`. Não é preciso implementar método algum. `Serializable` é simplesmente um flag que indica a possibilidade de converter o objeto em uma representação persistente.

```
public class Registro implements Serializable {
```

Os objetos são convertidos e identificados com um número de série (daí o nome), para garantir a segurança e evitar a corrupção dos dados. Vários métodos operam sobre objetos do tipo `Serializable`. No pacote `java.io`, as classes `ObjectOutputStream` e `ObjectInputStream` oferecem os métodos `writeObject()`, que recebe um objeto `Serializable` como argumento e `readObject`, que tenta ler um objeto a partir de um fluxo de dados e o retorna. Essas classes são subclasses de `FilterOutputStream` e `FilterInputStream` e podem ser concatenadas com qualquer `OutputStream` ou `InputStream`. O exemplo abaixo mostra como objetos podem ser armazenados e recuperados de um arquivo (`armario`). O tratamento das exceções foi omitido do trecho de código abaixo por clareza.


```
ObjectInputStream out =
     new ObjectOutputStream(new FileInputStream(armario));
Arco a = new Arco();
Flecha f = new Flecha();
out.writeObject(a);      // grava objeto Arco em armario
out.writeObject(f);      // grava objeto flecha em armario
(...)
ObjectInputStream in =
     new ObjectInputStream(new FileInputStream(armario));
Arco primeiro = (Arco)in.readObject(); // retorna Object (requer cast)
Flecha segundo = (Flecha)in.readObject();  // recupera os dois objetos
```

Alguns objetos não são serializáveis, devido à sua natureza mutante. `Streams`, por exemplo, não o são. Se um objeto serializável contém uma referência para um objeto não-serializável, *toda a operação* de serialização falha. Nestes casos, pode-se resolver o problema declarando que o objeto é transiente e não deve ser armazenado como parte do objeto. Isto é feito com a palavra-chave `transient`:

```
public class Registro implements Serializable {
 private transient InputStream in; // não será armazenado no processo
 private int numero;
...
```

Na seção seguinte veremos uma aplicação do `RandomAccessFile` na construção de um banco de dados.

Entrada e Saída Padrão

Java possui o conceito de uma entrada padrão, saída padrão e erro padrão. Esses objetos são definidos na classe `System` e são:

```
System.out   System.in   System.err
```

O erro padrão é semelhante à saída padrão, mas é utilizado para fornecer um fluxo alternativo de dados para mensagens de erro que podem ser filtradas através de uma redireção.

Como quase todo programa usa entrada e saída padrão, Java incluiu esses objetos na classe `System`, que pertence ao pacote `java.lang`, importada automaticamente em cada unidade de compilação. Quaisquer outras operações de entrada e saída requerem o uso da biblioteca `java.io`.

Algumas Classes para entrada de dados

InputStream e Reader

`InputStream` é classe abstrata e superclasse de todas as classes de entrada de dados. Em geral, só é utilizada para declarar classes que serão instanciadas via uma das suas especializações abaixo. `Reader` é classe que é superclasse de todas as classes de entrada de caracteres Unicode.

FileInputStream e FileReader

Operações básicas para ler um byte ou vetor de bytes de um arquivo. Os principais construtores são:

```
public FileInputStream(String name)
                       thows FileNotFoundException;
public FileInputStream(File file)
                       thows FileNotFoundException;
```

e os principais métodos:

```
public native void close() thorws IOExceoption;
public native int read() throws IOException;
public native int available() throws IOException;
```

Um exemplo de como instanciar um objeto `FileInputStream`:

```
FileInputStream fis;
File arq;
try {
arq = new File("teste.txt");
fis = new FileInputStream(arq);
  (...)
}
catch (FileNotFoundException fnfe) { ... }
```

SequenceInputStream

Permite a concatenação de dados de um ou mais fluxos de entrada de dados. Veja uma aplicação no exemplo "cat" abaixo.

Principal Construtor:

```
public SequenceInputStream(InputStream s1, InputStream s2);
```

DataInputStream

Permite a leitura de tipos primitivos de uma maneira independente de plataforma.

Construtor:

```
public DataInputStream(InputStream in);
```

Principais Métodos:

```
public final String readInt() throws IOException;
public final String readFloat() throws IOException;
public final String readDouble () throws IOException;
```

BufferedReader

Permite a leitura de texto de uma maneira independente de plataforma.

Construtor:

```
public BufferedReader(Reader in);
```

Principal Método:

```
public final String readLine() throws IOException;
```

O exemplo a seguir é um programa semelhante ao "cat" do Unix. Ele concatena arquivos cujos nomes recebe como argumento. Caso não receba argumentos, lê a entrada padrão. É um exemplo do uso de `FileInputStream`, `SequenceInputStream` e `BufferedReader`.

```
/** Simple Unix-like con-CAT-enate program.
 *
 *
 */

import java.io.*;

public class cat {

    // declarações de variáveis estáticas
    static FileInputStream fis;
    static BufferedReader dis;
    static PrintStream o = System.out;

    static final int EOF = -1;


    public static void main (String args[]) {
        int c;
        String s;
        SequenceInputStream sis = null;
```



```
            try {
                // havendo args. na linha de comando...
                for (int i = (args.length - 1); i>=0; i--) {
                    // cria novo descritor
                    File f = new File(args[i]);

                    fis = new FileInputStream(f);
                    sis = new SequenceInputStream(fis, sis);
                }
                // lê cada caractere de sis e imprime
                while((c = sis.read()) != EOF) {
                    o.print((char)c);
                }
            }
            catch (FileNotFoundException fnfe) {
                o.println("Eeeeek!!! File Not Found!");
            }
            catch (IOException ioe) {
                o.println("Ooops... An IO Exception");
            }
            // erro se não houver argumentos...
            catch (NullPointerException npe) {
                try {
                    // lê entrada padrão
                    dis = new BufferedReader(new InputStreamReader(System.in));
                    while((s = dis.readLine()) != null) {
                        o.println(s);
                    }
                }
                catch (IOException ioe){
                    o.println("Arghh!!! IO Exception");
                }
            }
        }
    }
```

Algumas classes para saída de dados

OutputStream e Writer

Classe abstratas. Superclasses de todas as classes de saída de dados e de caracteres, respectivamente. Em geral, só são utilizadas para declarar objetos que serão instanciados via uma das suas especializações abaixo.

FileOutputStream e FileWriter

Escreve dados em um arquivo especificado por nome ou por um objeto descritor de arquivo.

Principais Construtores:

```
public FileOutputStream(String name)
                        throws IOException;
public FileOutpútStream(File file)
                        throws IOException;
```

Principais Métodos:

```
public void close() throws IOException;
protected void finalize() throws IOException;
```

PrintStream

Implementa vários métodos para impressão de representações textuais de tipos primitivos Java. É a classe mais usada para impressão de texto. A variável `System.out` é um `PrintStream`.

Principais Métodos

```
public synchronized void print(String s);
public synchronized void print(char[] s);
public void println();
public synchronized void println(String s);
public void flush();
public void close();
```

DataOutputStream

Permite a escrita de texto e tipos primitivos de uma forma independente de plataforma.

Construtor:

```
public DataOutputStream(OutputStream out);
```

Principais Métodos:

```
public final int size();
public final void writeBytes(String s)
                             throws IOException;
```


Exercícios

1. Crie uma aplicação que leia um arquivo do disco e extraia todos os descritores HTML dele. Descritores HTML começam em "<" e terminam em ">".
2. Crie um programa que pede seu nome e endereço e o salva em um arquivo.
3. Crie um programa que leia os nomes e endereços já digitados a partir do arquivo criado.
4. Crie um programa que use `ObjectOutputStream` e `ObjectInputStream` para serializar e salvar objetos em disco. Como objeto, salve uma cor.

6.3. Aplicação de banco de dados usando RandomAccessFile

Nesta seção, descreveremos o desenvolvimento de uma aplicação de banco de dados simples.

O programa oferecerá uma interface para um arquivo onde serão armazenados registros e o usuário poderá controlar a criação de novos registros, administrar os registros existentes, listar todos os registros já armazenados ou recuperá-los individualmente.

Os arquivos e programas discutidos a seguir estão no disquete que acompanham este trabalho (`jad/apps/`).

Como o objetivo deste capítulo é entrada e saída em Java, não nos concentraremos em detalhes relativos ao desenvolvimento da interface gráfica do usuário ou em outros detalhes básicos da linguagem Java. Por este motivo, utilizaremos a interface do usuário orientada a caracter para mostrar o funcionamento do banco de dados. As várias outras interfaces disponíveis (gráfica, applet) são totalmente compatíveis já que operam sobre o mesmo modelo de dados.

Projeto da aplicação

A aplicação terá três camadas lógicas independentes que são:

- Camada de apresentação, consistindo da interface do usuário
- Camada do domínio da aplicação, consistindo das classes que representam conceitos como 'registro' e 'banco de dados'; e serviços, como 'driver para banco de dados em arquivo de acesso aleatório'.

- Camada de armazenamento, consistindo de um arquivo do sistema.

O objetivo desta seção é implementar uma subcamada de serviços que permita manipular um banco de dados em um arquivo de acesso aleatório. Adicionalmente, mostraremos a aplicação dessa classe em uma interface do usuário orientada a caracter.

Implementação do banco de dados

Para implementar o banco de dados precisamos implementar a interface `bancodados.BancoDados` (veja mais informações sobre a aplicação na documentação HTML que a acompanha). Cada método deve realizar suas operações sobre um `RandomAccessFile` que armazena objetos do tipo `Registro` em disco. Como não há métodos `readObject()` e `writeObject()` em `RandomAccessFile`, precisamos decompor o registro em suas partes. Precisamos portanto tomar decisões quanto à organização das partes de cada registro além da organização dos registros individuais no banco de dados.

Decidimos organizar (armazenar) cada registro no arquivo da seguinte forma (nesta ordem):

- **`int`**: número do anuncio
- **`String`**: texto do anuncio
- **`long`**: data (tempo em milissegundos desde 1/1/1970)
- **`String`**: autor do anuncio

Podemos usar os métodos `writeInt()`, `writeLong()` e `writeUTF()` para gravar os tipos `int`, `long` e `String`, respectivamente e `readInt()`, `readLong()` e `readUTF()` para recuperá-los posteriormente.

Quanto à organização dos registros no banco de dados decidimos que:

- Os registros serão acrescentados ao arquivo em seqüência.
- Cada novo registro será acrescentado no final do arquivo com um número igual ao maior número pertencente a um registro existente mais um, ou `100`, se não houver registros;
- Registros removidos terão o seu número alterado para `-1` (continuarão ocupando espaço no arquivo).
- Registros alterados serão primeiro removidos e depois acrescentados no final do arquivo com o mesmo número que tinham antes (também continuarão ocupando espaço no arquivo).

A classe que desenvolvemos está em `bancodados/tier2/local/` e chama-se `BancoDadosArquivo.java`. Implementa `BancoDados` podendo ser utilizada por qualquer outra classe que manipule com a interface. Precisa portanto implementar cada método de `BancoDados`. A classe possui um objeto `RandomAccessFile` que representa o arquivo onde os dados serão armazenados. Suas variáveis membro e a implementação de seu construtor estão mostrados abaixo:

```
public class BancoDadosArquivo implements BancoDados {

   private RandomAccessFile arquivo;   // descritor de arquivo
   private boolean arquivoAberto;      // inicialmente false
   private Hashtable bancoDados;       // relaciona posicao do ponteiro
                                       // do RandomAccessFile com registro
   private int maiorNumReg = 0;        // Maior número de registro

   public BancoDadosArquivo(String arquivoDados) throws IOException {
      try {
         arquivo = new RandomAccessFile(arquivoDados, "rw");
         arquivoAberto = true;
      } catch (IOException e) {
         close();
         throw e;    // propaga excecao para metodo invocador
      }
   }
(...)
```

A referência `arquivo` é utilizada em todos os métodos. Abaixo listamos os métodos `addRegistro()`, que adiciona um novo registro e `getRegistros()` que recupera todos os registros e os retorna.

```
public synchronized void addRegistro(String anuncio, String contato) {
      try {
         arquivo.seek(arquivo.length());  // posiciona ponteiro no fim
         arquivo.writeInt(getProximoNumeroLivre());
         arquivo.writeUTF(anuncio);
         arquivo.writeLong(new Date().getTime());
         arquivo.writeUTF(contato);
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }
   }



public Registro[] getRegistros() {
```



```
      try {
        arquivo.seek(0);        // posiciona ponteiro no inicio do arquivo
        bancoDados = new Hashtable();
        Vector regsVec = new Vector();
        while (temMais()) {
           long posicao = arquivo.getFilePointer();
           int numero = arquivo.readInt();           // Lê proximo int
           String anuncio = arquivo.readUTF();       // Lê próximo String
           Date data = new Date(arquivo.readLong()); // Le proximo long
           String contato = arquivo.readUTF();
           maiorNumReg = Math.max(numero, maiorNumReg); // guarda maior reg
           if (numero < 0) continue;    // flag p/ registro removido é -1
           Registro registro = new Registro(numero, anuncio, data, contato);
           bancoDados.put(new Integer(numero), posicao + "");
           regsVec.addElement(registro);
        }
        Registro[] regs = new Registro[regsVec.size()];
        regsVec.copyInto(regs);      // copia vector em vetor[]
        return ordenar(regs);        // retorna vetor ordenado
      } catch (IOException ioe) { // (...)
      }
    }
```

Para remover um registro, é preciso saber em que posição ele está. O `Hashtable bancoDados` (definido em `getRegistros()`) contém um mapa que relaciona o número do registro com a posição no arquivo. O método `removeRegistro()` utiliza esta informação para localizar o registro que ele deve marcar como removido.

```
  public synchronized boolean removeRegistro(int numero)
     throws RegistroInexistente {
        try {
          getRegistros();
          String pointer = (String)bancoDados.get(new Integer(numero));
          if (pointer == null)
             throw new RegistroInexistente("Registro não encontrado");
          long posicao = Long.parseLong(pointer);
          arquivo.seek(posicao);
          int numReg = arquivo.readInt();
          if (numReg != numero)
             throw new RegistroInexistente("Registro não localizado");
          arquivo.seek(posicao);
          arquivo.writeInt(-1);   // marca o registro como removido
          arquivo.seek(0);
        } catch (IOException ioe) { // (...)
        }
        return true;
```



```
}
```

Nesta interface que desenvolvemos para o arquivo usando `RandomAccessFile`, os registros removidos nunca são realmente removidos. Para limpar o arquivo, livrando-o de espaço ocupado inutilmente, é preciso exportar todos os registros válidos e importá-los de volta em um novo arquivo.

Um outro modelo, mais eficiente para um banco de dados baseado no `RandomAccessFile` é proposto em [JavaWorld98]. Em uma versão futura, esta aplicação pode se tornar mais eficiente implementando um banco de dados mais eficiente. Desde que a nova implementação tenha a interface `BancoDados`, poderemos substituir a antiga versão pela nova e não será preciso alterar uma linha sequer no restante da aplicação.

Desenvolvimento da interface do usuário

Os arquivos utilizados nesta aplicação estão nos subdiretórios a seguir. Em negrito está o único arquivo que trata da interface com o usuário:

Subdiretório	Arquivo-fonte Java	Conteúdo
`bancodados/`	**`DadosClientTextUI.java`**	interface do usuário orientada a caracter
`bancodados/server`	`BancoDados.java`	interface genérica para o banco de dados (interface)
`bancodados/server`	`Registro.java`	representação de um registro (classe concreta)
`bancodados/server/arquivo`	`BancoDadosArquivo.java`	implementação de `BancoDados`

A interface do usuário deve manipular com um objeto `BancoDados`. Na prática, estará manipulando com o `RandomAccessFile` através da classe `BancoDadosArquivo` mas ela não precisa saber disso.

A classe `DadosClientTextUI` declara uma variável membro do tipo `BancoDados`:

```
private BancoDados client;   // cliente indep. de interface do usuario
                             // e da forma de armazenamento
```


e em todos os seus métodos invoca métodos de `client`. Apenas o menu principal refere-se ao `BancoDadosArquivo`, para instanciá-lo e passar sua referência para `client`. A partir daí, todos os métodos operam sobre a interface `BancoDados`.

Se o usuário decidir criar um novo registro, por exemplo, a aplicação chamará o método local `criar()`, que contém:

```
public void criar() throws IOException {
      BufferedReader br;
      br = new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Texto: ");
      System.out.flush();
      String texto = br.readLine();
      System.out.print("Autor/Endereco: ");
      System.out.flush();
      String autor = br.readLine();
      client.addRegistro(texto, autor); // método de BancoDados
   }
```

Para listar todos os registros, o método `mostrarTodos()` é chamado:

```
public void mostrarTodos() throws IOException, RegistroInexistente {

      BufferedReader br;
      br = new BufferedReader(new InputStreamReader(System.in));

      Registro[] regs = client.getRegistros();
      System.out.println("----- Registros armazenados -----");
      for (int i = 0; i < regs.length; i++) {
         mostrar(regs[i]);
         if (i % 3 == 0) {
            System.out.print("Aperte qualquer tecla para continuar...");
            System.out.flush();
            br.readLine();
         }
      }
      System.out.println("\n----- Listados " + regs.length + " registros -----")
   }
```

Em nenhum dos métodos há indicações que acontece alguma coisa em um `RandomAccessFile`. Sendo assim, podemos substituir facilmente o banco de dados `BancoDadosArquivo` por outra implementação. Faremos isto no próximo módulo.

Glossário

Streams – fluxos unidirecionais de informação. Pode ser ler informações que chegam em pedaços através de um *stream*. Pode-se enviar informações através de um *stream* que será montada no destino.

Serialização – transformação de objetos em vetores (*arrays*) de bytes marcados com um número de série e outros dispositivos para garantir a segurança e consistência dos dados.

Persistência – capacidade de um objeto preservar seu estado de forma que possa ser recuperado posteriormente mesmo quando o objeto que o criou não mais existir.

Cliente – parte de uma aplicação que utiliza serviços oferecidos por outra parte da mesma aplicação ou por outra aplicação.

Servidor – parte de uma aplicação que oferece um serviço usado por clientes.

6.4. Soquetes e datagramas TCP/IP

Uma das interfaces do usuário que podemos usar para acessar a aplicação de banco de dados é a interface baseada em applet. A interface Applet é bem parecida com a interface gráfica standalone. Utiliza o mesmo painel de controles (proporcionado pela classe `DadosClientPanel`). Porém, devido à restrições impostas aos applets, só temos acesso aos bancos de dados localizados na mesma máquina onde está o servidor HTTP que nos serviu o applet. Outra desvantagem de utilizar a aplicação de banco de dados na Web via applet é a quantidade de software que terá que ser transferido para o browser para que a aplicação funcione. Para ter uma acesso direto, via JDBC, o cliente teria que antes descarregar:

- As classes do pacote bancodados: `Registro`, `RegistroInexistente` e `BancoDados`
- As classes da camada de apresentação: `DadosClientPanel` e `DadosApplet`
- A classe `BancoDadosJDBC`
- Os drivers JDBC utilizados para realizar a conexão
- Dependendo do tipo de driver utilizado (se for do tipo 1 ou 2), software adicional (dependente de plataforma) poderá ser necessário

- Todo o pacote `java.sql` se o browser não suportar JDBC (browsers Netscape e Internet Explorer versões 3 e anteriores)

Todas as classes acima podem ser empacotadas e comprimidas em um arquivo JAR (Java Archive) para transferência mais eficiente pela rede mas tudo isso pode ser inútil se o usuário decidir conectar-se a um arquivo, pois estará sujeito a novas restrições.

Usando um servidor intermediário, para receber requisições do cliente (independentes de tecnologia de banco de dados), podemos fornecer um applet utilizável tanto para acesso a arquivos como para acesso a bancos de dados JDBC. O cliente teria apenas as classes:

- As classes do pacote `bancodados`: `Registro`, `RegistroInexistente` e `BancoDados`
- As classes da camada de apresentação: `DadosClientPanel` e `DadosApplet`
- Uma classe cliente para o serviço remoto, que implemente os métodos em `BancoDados`: `TCPClient`

Fundamentos de Redes e Fluxos de Dados

Na API Java, a classe `InetAddress` representa um endereço IP. Pode-se obter o endereço usando seu método `getAddress()`. Pode-se tentar obter o nome da máquina usando o método `getHostName()`, que consultará o serviço de nomes para descobrir o nome correspondente ao endereço.

Um cliente em geral não deseja se comunicar com uma máquina mas com um determinado serviço em uma máquina. Os serviços em uma máquina servidora são localizáveis através de uma porta identificados por um número. Toda a comunicação em rede é fundamentada no esquema máquina/porta, que constitui um soquete. Na API Java, os soquetes são representados por várias classes, dependendo do seu tipo. A comunicação via protocolo TCP (*Transfer Control Protocol*), confiável, é suportada pelas classes `Socket` (soquete de dados) e `ServerSocket` (soquete do servidor). A comunicação via UDP (*Unreliable Datagram Protocol*), não-confiável, é suportada pelas classes `DatagramSocket` (soquete de dados UDP), `DatagramPacket` (pacote UDP) e `MulticastSocket` (soquete UDP para difusão).

Depois que um soquete é criado, pode-se obter, a partir dele, fluxos de dados ou de caracteres. Esses fluxos podem então ser tratados da maneira

convencional, concatenando-os a outros fluxos para filtrar dados até se obter um formato adequado para tratamento dentro do programa.

Toda a comunicação TCP/IP pode ser realizada usando as classes mencionadas acima. O pacote `java.net`, porém, traz também outras classes que oferecem suporte a protocolos de transferência de dados. A URL (Uniform Resource Locator) é representada pela classe `URL`. Pode ser construída a partir de uma string contendo a URL. Tendo um objeto URL, pode-se usar seus métodos `getStream()` para obter um fluxo de dados a partir da URL ou seu método `getContent()` para ler um objeto contendo os dados. Se o tipo de dados for suportado, será possível manipulá-lo usando os recursos da linguagem Java. Ser não for, pode-se criar manuseadores de conteúdo para tipos não suportados usando as classes `ContentHandler` e `ContentHandlerFactory`, também disponíveis no pacote `java.net`.

Recursos transferidos pela rede são recebidos em forma de fluxo de bytes e geralmente precisam de um cabeçalho de meta-informação para identificar o tipo de dados que está sendo transferido. A classe `URLConnection` permite que a informação de cabeçalho da conexão seja recuperada. Com ela pode-se descobrir o comprimento dos dados, se os dados são uma página HTML, um arquivo executável, uma imagem JPEG ou outro tipo de dados através do seu tipo MIME (Multipart Internet Mail Extensions).

Pode-se usar tanto soquetes quanto URLs para se obter os mesmos resultados. A primeira alternativa é mais complexa. Para usá-la, precisamos ter todos os agentes concordando acerca de um determinado protocolo e implementar toda a comunicação de meta-dados para informar sobre tipos de conteúdo. Usando `URLs` e `URLConnection`, podemos ter um desenvolvimento mais simples, mas, por outro lado, temos que lidar com o *overhead* extra imposto pelos protocolos de transferência de dados.

Soquetes TCP

Soquetes TCP são usados na maioria das aplicações IP que necessitam de garantia de recebimento de pacotes e ordem em que os pacotes são recebidos. Para criar um soquete de dados em Java para enviar dados para um servidor HTTP na máquina info.acme.com, pode-se fazer:

```
InetAddress end = InetAddress.getByName("info.acme.com");
```



```
Socket con = new Socket(end, 80);
InputStream dados = con.getInputStream();
OutputStream comandos = con.getOutputStream();
// enviar comandos e receber dados do processo remoto...
```

Esta é apenas uma das formas de construir um soquete. Há outros construtores mas todos recebem um endereço para a máquina servidora e uma porta de serviço.

As classes `InputStream` e `OutputStream` lidam com os dados como bytes. Possuem métodos para ler e gravar bytes e vetores de bytes. As classes `Reader` e `Writer` transmitem e recebem caracteres Unicode de 16 bits.

Quando um `Socket` ou `Process` é criado, pode-se ler e gravar dados nele em forma de bytes obtendo fluxos de bytes de entrada e saída através dos métodos `getOutputStream()` e `getInputStream()`. Para ler ou gravar caracteres ao invés de bytes, pode-se usar um dos fluxos de conversão entre bytes e caracteres: `InputStreamReader` e `OutputStreamWriter`:

```
Socket con = new Socket("maquina.com.br", 4444);
Reader = new InputStreamReader(con.getInputStream());
Writer = new OutputStreamWriter(con.getOutputStream());
```

Para construir um servidor que fique esperando a conexão de clientes, pode-se usar a classe `ServerSocket`. O soquete de servidor só é necessário enquanto não se obtém um soquete de dados, retornado por seu método `accept()`, que bloqueia a execução do *thread* até que um soquete de dados seja recebido. Depois de obtida a conexão de dados, o soquete do servidor pode ser descartado ou colocado novamente para escutar outro cliente. Suponha que na máquina info.acme.com haja um servidor Web escrito em Java. O código do servidor pode ter algo como:

```
ServerSocket escuta = new ServerSocket(80);
while(true) {
Socket cliente = escuta.accept();  // espera aqui por cliente
InputStream comandos = cliente.getInputStream();
OutputStream dados = cliente.getOutputStream();
// receber comandos e enviar dados ao processo remoto...
```

No exemplo acima, o servidor só poderá lidar com um novo cliente depois que o cliente atual tiver terminado seu trabalho. Para que um servidor possa trabalhar com múltiplos clientes ao mesmo tempo, terá que passar os soquetes de

dados para novos threads, mantendo o thread principal exclusivamente para escutar e receber novos clientes.

Soquetes UDP

Quando não há necessidade de se garantir que um pacote chegará ao seu destino ou quando a ordem dos pacotes não interessar, pode-se optar pelo uso do protocolo UDP como uma forma mais eficiente e rápida de transmissão de dados. Um típico exemplo é na transmissão de áudio e vídeo onde o *overhead* da transmissão TCP é inaceitável e não faz sentido retransmitir um pacote que não conseguiu chegar ao seu destino.

Para que dois agentes possam se comunicar usando uma conexão UDP, é preciso que ambos tenham um `DatagramSocket` conectado a uma porta de suas máquinas. Isto pode ser feito da forma:

```
DatagramSocket udp = new DatagramSocket();
```

No exemplo acima, não foi especificada uma porta e o sistema utilizará uma porta qualquer que estiver disponível. É possível especificar o número da porta usando outro construtor que recebe a porta como argumento ou obter a porta fornecida pelo sistema usando o método `getLocalPort()`. A comunicação só é possível os agentes tiverem a porta e a máquina do destinatário. Esta porta pode ser uma porta conhecida ou pode ser transmitida via um soquete de outra conexão.

Para enviar dados, é preciso construir um `DatagramPacket` com os dados a serem enviados e usar o método `send()` do `DatagramSocket`. O `DatagramPacket` também contém o endereço do destinatário e a porta onde ele espera pelo pacote:

```
DatagramSocket udp = new DatagramSocket();
byte[] info = {'o', 'i'};
int portaDestino = 3333;
InetAddress destino = InetAddress.getByName("longe.com");
DatagramPacket pacote = new DatagramPacket(info, info.length, destino, porta);
udp.send(pacote);
```

O processo remoto pode recuperar o pacote usando o método `receive()`. O pacote recebido conterá, além dos dados, o endereço e porta do remetente que podem ser recuperados através dos métodos `getAddress()` e `getPort()`.

```
byte info = new byte[256];
```



```
DatagramSocket udp = new DatagramSocket(3333);
DatagramPacket pacote = new DatagramPacket(info, info.length);
udp.receive(pacote);
InetAddress remetente = udp.getAddress();
int portaRemetente = udp.getPort();
byte[] dados = udp.getData();
```

Multicasting

Multicasting é uma forma de enviar informações a vários agentes ao mesmo tempo, de forma análoga a transmissões de rádio e TV. O agente transmissor cria um canal baseado em uma máquina e porta e transmite informações que são captadas em forma de pacotes UDP por todos os receptores sintonizados. Este recurso tem várias aplicações não só na difusão de áudio e vídeo mas também em ferramentas de colaboração, sincronização de servidores, balanceamento de cargas, etc.

É possível construir um canal de difusão usando a classe `MulticastSocket`. Esta classe é uma subclasse de `DatagramSocket` e, portanto, herda os seus métodos `send()` e `receive()`. Depois de criado o `MulticastSocket`, basta usar `send()` e `receive()` para enviar ou receber dados através do canal de difusão.

```
MulticastSocket grupo = new MulticastSocket();
InetAddress fonte = InetAddress.getByName("224.0.0.1");
byte[] info = {'S', 'O', 'S'};
DatagramPacket dados = new DatagramPacket(info, info.length, fonte, 5555);
grupo.send(dados);
```

Para receber dados através de um canal de difusão, é preciso sintonizá-lo usando o método `joinGroup()`. Depois de conectado, pode-se enviar ou receber dados. Para parar de receber e enviar dados, deixa-se o grupo usando o método `leaveGroup()`.

```
MulticastSocket grupo = new MulticastSocket();
InetAddress fonte = InetAddress.getByName("224.0.0.1");
grupo.joinGroup(fonte);          // entra no grupo  (sintoniza)
byte[] info =  new byte[];
DatagramPacket dados = new DatagramPacket(info, info.length);
grupo.receive(dados);            // recebe dados
byte recebido = dados.getData();
grupo.leaveGroup(fonte);         // sai do grupo
```


6.5. Construção de uma aplicação distribuída

A proposta desta seção é incluir uma camada adicional entre o cliente e o servidor nas aplicações apresentadas na seção anterior. Mostraremos a construção de uma aplicação simples usada para medir a transferência de bytes e a extensão da aplicação de banco de dados (seção anterior) para suportar uma camada adicional baseada em protocolos TCP/IP.

Aplicação de transferência de bytes

A aplicação TCP/IP envolve a definição de um cliente, um servidor e um protocolo. Nesta aplicação, o cliente utilizará o protocolo para enviar comandos e dados para o servidor que, estará no ar aguardando clientes. Quando um cliente tentar conectar-se ao servidor, localizando-o através da máquina onde reside e da sua porta de serviços, o servidor aceita a conexão e cria um thread para lidar com este cliente em um novo soquete de dados, liberando o thread principal para que possa aguardar novos clientes. O thread de dados então deve obter um fluxo de dados do cliente, analisar as informações recebidas e identificar o comando do protocolo correspondente. Conhecendo o comando, o servidor tomará decisões sobre o que fazer: aguardar mais dados, imprimir estatísticas ou fechar a conexão.

O protocolo é definido através de uma interface Java, que deve estar presente tanto no servidor quanto no cliente. Ela define constantes que identificam comandos e a assinatura dos métodos chamados no servidor:

```
package bench.server.tcpip;
import java.io.IOException;

public interface DataMon {

   public static final int SEND  = 100;
   public static final int CLOSE = 200;
   public static final int PRINT = 300;

   public void envia(long inicio, int tamanho, byte[] bytes)
                     throws IOException;
   public void close() throws IOException;
   public void print() throws IOException;

}
```

O servidor consiste das classes `bench.ServerFrame`, `bench.server.tcpip.Server` e `DataMonImpl`. A classe `ServerFrame` apenas

proporciona a interface gráfica e redirecionamento de saída. A inicialização do servidor ocorre no método `init()` da classe `Server`, que cria um `ServerSocket` (soquete de serviços) na porta PORT e inicia o servidor em um thread próprio para aguardar e receber clientes (`server`).

```
public class Server extends ServerFrame {

   private static final int PORT = 1999;
   private static int numero = 0;

   private ServerSocket serv = null;
   private Socket client = null;
   private Thread server = null;

   public void init() {
       try{
            serv = new ServerSocket(PORT);
            server = new ServerThread();
            server.start();
        } catch (java.io.IOException e) {/* (...) */}
    }
```

O thread do servidor foi implementado como uma classe interna (`ServerThread`) que possui um loop que fica sempre aguardando clientes. Quando um cliente aparece, um novo objeto `DataMonImpl` é criado, recebendo o soquete de dados do cliente (`client`) como argumento. Um novo thread (`dataThread`) também é criado, recebendo como argumento o objeto `DataMonImpl` (`dataRef`). Em seguida é iniciado, causando a execução do método `run()` em `DataMonImpl`.

```
/** Thread do servidor. Classe interna. */
private class ServerThread extends Thread {

   public void run() {
     try {
        System.out.println ("Iniciando servidor...");
        DataMonImpl dataRef = null;
        while (server == Thread.currentThread()) {
           System.out.println("Aguardando clientes na porta " + PORT + "!");
           client = serv.accept(); // AGUARDA CLIENTES AQUI!
           System.out.println("Cliente conectado!");

           // cria objeto
           dataRef = new DataMonImpl(client);
           client = null;
```



```
                Thread dataThread = new Thread(dataRef);   // thread dados
                dataThread.start();
                try {
                   Thread.sleep(100);
                } catch (InterruptedException e) {}
             }
          }  // (... catch e finally ...)
       } // fim do run()
    }
(...)
```

Cada conexão de dados individual é tratada pelo objeto `DataMonImpl` criado por `Server` no thread de dados. O objeto do tipo `DataMonImpl` recebe o soquete de dados e extrai seu fluxo de dados de entrada:

```
public class DataMonImpl implements DataMon, Runnable {

    private DataInputStream in;
    private Hashtable table;
    private Socket client;
    private int last = 0;

    /** Construtor
     */
    public DataMonImpl(Socket con) throws IOException {
        client = con;
        table = new Hashtable(10);
        in = new DataInputStream(con.getInputStream());
    }
```

Seu método `run()` redireciona o controle para um dos métodos da interface `DataMon`, implementada nesta classe, de acordo com o comando recebido pelo cliente (veja código no disquete).

O cliente TCP/IP está embutido nas classes `bench.client.tcpip.Sender` e `Client`. A classe `Client` inicializa a conexão, criando um soquete e tentando conectar no serviço remoto. Obtém um fluxo de dados de saída a partir do soquete e o passa para um objeto `Sender`, cuja função é utilizar o fluxo de dados para enviar informações para o servidor:

```
public class Client extends ClientFrame {

    private static final int    PORT = 1999;
    private static final String host = "localhost";

    private DataMon dataRef;  // referencia bench.server.tcpip.Deposito
    (...)
```



```
public void connect() {
    try {
        // tenta realizar conexão ao servidor...
        Socket con = new Socket(host, PORT);

        // obtem outputstream do socket...
        OutputStream out = con.getOutputStream();

        // cria objeto adaptador (adapter)
        dataRef = new Sender(out);
    } catch (Exception e) { /* (...) */ }
}
```

A classe `Sender` ignora que está enviando dados pela rede. Simplesmente utiliza os fluxos de dados e envia os dados através deles. Veja o método `envia()` no disquete.

Aplicação de banco de dados

Nesta seção, mostraremos uma forma de acrescentar uma camada intermediária na aplicação de banco de dados (discutida na seção anterior). Todas as classes (arquivo-fonte `.java` inclusive) podem ser encontradas nos subdiretórios `bancodados/tier3/tcpip` e `bancodados/tier2/remote`.

O objetivo é desenvolver um cliente, um servidor e um protocolo que possam ser conectados à aplicação de banco de dados, permitindo que ela possa ser usada em rede. Para que a interface do usuário possa interagir com o servidor com a mesma facilidade com que interagia com o arquivo de banco de dados (seção anterior), é preciso que o módulo "cliente" da aplicação distribuída implemente a interface `bancodados.BancoDados`, que contém todos os métodos que as interfaces do usuário utilizarão para manipular com o banco de dados.

Cliente e BDProtocol

O cliente deve utilizar um `Socket` para conectar-se a um servidor. Conseguindo, deve obter os fluxos de dados de entrada e saída deste soquete, já que estaremos enviando e recebendo dados. As instruções enviadas para o cliente serão strings e tipos primitivos, portanto utilizaremos um filtro do tipo `DataOutputStream`, para o fluxo de saída. O servidor deve retornar objetos (registros) e tipos primitivos encapsulados em objetos, portanto, filtraremos os dados de entrada em um `ObjectInputStream`. A definição da classe `TCPClient` (em `bancodados.tier2.remote`) e seu construtor estão mostrados abaixo:


```
public class TCPClient implements BancoDados {

   public static final int PORT = 1729;
   private DataOutputStream out;
   private ObjectInputStream in;
   private Socket con;

   public TCPClient(String hostname) throws IOException {
      try {
         con = new Socket(hostname, PORT);
         out = new DataOutputStream(con.getOutputStream());
         in =  new ObjectInputStream(con.getInputStream());
      } catch (UnknownHostException e) {
         throw new IOException(e.toString());
      }
   }
```

A classe TCPClient pode ser usada em qualquer interface do usuário, solicitando-se ao usuário o endereço (DNS) da máquina onde roda o servidor:

```
    String host = ...;
BancoDados bd = new TCPClient(host);
(...)
Registros regs[] = bd.getRegistros();
```

O restante da classe `TCPClient` lida com a implementação dos métodos de `bancodados.BancoDados`. Os métodos lidam com fluxos de entrada e saída, ignorando sua origem (a rede). Os comandos enviados para o servidor têm que ser suportados por um protocolo que chamamos de `BDProtocol`. Os comandos de `BDProtocol` são:

```
length
getregistro <int>
addregistro <String UTF> <String UTF>
setregistro <int> <String UTF> <String UTF>
remregistro <int>
search <String UTF>
getall
exit
```

Os valores entre "<...>" acima representam argumentos que devem ser passados quando o comando correspondente for emitido pelo cliente. Os comandos são enviados usando métodos de `DataOuputStream` como:

`writeChars()` (seqüência de caracteres), para o nome do comando,

`writeInt()`, para argumentos inteiros,

`writeUTF()`, para argumentos do tipo `String`.

Veja a implementação do método de `BancoDados setRegistro()` abaixo. Depois que todos os dados são enviados, o thread bloqueia aguardando o término de `readObject()`:

```
public boolean setRegistro(int numero, String texto, String autor)
       throws RegistroInexistente {
      try {
         out.writeChars("setRegistro ");
         out.writeInt(numero);
         out.writeUTF(texto);
         out.writeUTF(autor);
         Object resultado = in.readObject(); // retornado pelo servidor
         if (resultado instanceof Boolean) {
            return ((Boolean)resultado).booleanValue();
         } else {
            throw new RegistroInexistente((String)resultado);
         }
      } catch (Exception e) { /** (...) */}
   }
```

Servidor

O servidor será executado por uma aplicação na máquina remota, criada especialmente para montar o serviço de acesso a banco de dados. O programa terá que aguardar a conexão de um cliente. Obtendo-a, esperará por comandos e os decodificará.

As classes que implementam o servidor são várias. Estão no pacote `bancodados.tier3.tcpip`. O núcleo do servidor baseia-se na classe `BancoDadosImpl`. Esta classe exerce um papel duplo[1]: é uma implementação da interface `BancoDados`, agindo como um adaptador para os métodos em alguma outra implementação de `BancoDados` (uma fonte de dados local `BancoDadosJDBC` ou `BancoDadosArquivo`) e, é responsável pelo tratamento de dados através do soquete recebido de um cliente.

```
public class BancoDadosImpl extends TCPServer implements BancoDados {

   private BancoDados fonte;       // acesso a operacoes de fonte local
   private ComandoInputStream in = null;
   private ObjectOutputStream out = null;

   public BancoDadosImpl(BancoDados fonte)  {
      this.fonte = fonte;
```

[1] Isto obviamente não reflete um bom design orientado a objetos.


```
    }
(...)
```

A implementação de `BancoDados` simplesmente repassa a invocação de cada método para a fonte de dados passada no construtor. Todos os métodos têm a mesma forma:

```
(...)
public void addRegistro(String texto, String autor) {
      fonte.addRegistro(texto, autor);
}

public boolean setRegistro(int numero, String texto, String autor)
   throws RegistroInexistente  {
      fonte.setRegistro(numero, texto, autor);
      return true;
}
(...)
```

A classe `TCPServer`, estendida por `BancoDadosImpl`, é uma classe abstrata. Seu único método abstrato é:

```
public abstract void run(Socket data);
```

que é implementado em `BancoDadosImpl` onde os fluxos de entrada e saída do soquete data são obtidos, e toda a comunicação com o cliente que criou o soquete é realizada.

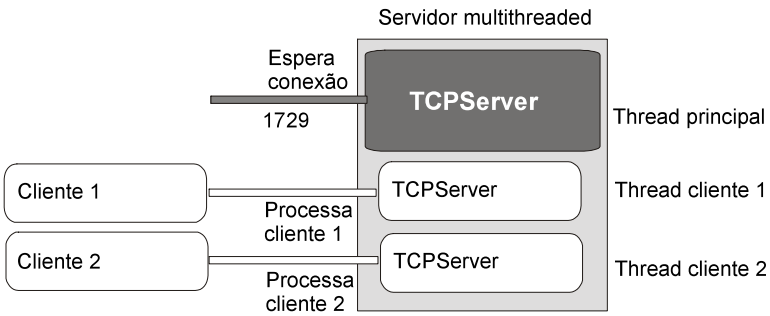


Figura 6-5

`TCPServer` implementa a parte do servidor que recebe os clientes. Inicia criando um thread principal que espera em loop até que a conexão solicitada por um cliente seja aceita. Toda a ação ocorre no seu método `run()`, que no thread principal aguarda os novos clientes. Quando a conexão é aceita, `TCPServer` cria uma cópia de si mesmo e inicia um novo thread na cópia. `run()` é executado novamente no clone mas desta vez o controle cai na uma segunda parte do método e é redirecionado para o método `run(Socket dados)`, implementado na sua subclasse `BancoDadosImpl`.

```
public abstract class TCPServer implements Cloneable, Runnable {
```



```
   private static int clientes = 0;     // número de clientes criados
   private static ServerSocket server;  // servico
   private boolean isDataConnection = false;    // nao é conexão de dados
   private Thread runner;      // thread que executará este objeto
   private Socket data;        // conexao de dados deste objeto
(...)
   public void run() {
      if (!isDataConnection) {   // se este eh o thread do servidor
         while(runner == Thread.currentThread()) {
            try {
               Socket dataSocket = server.accept(); // espera conexao dados
               System.out.println("Cliente conectado");
               TCPServer newSocket = (TCPServer)clone();  // Cria copia
               newSocket.isDataConnection = true; // flag nao eh server
               newSocket.data = dataSocket;  // socket passado adiante
               newSocket.runner = new Thread(newSocket);
               newSocket.runner.start(); // chama run() e cai no "else"
            } catch (...)
         }
      } else {    // se this.isDataConnection = true
         while(runner == Thread.currentThread()) {
            run(data);
         }
      }
   }
   public abstract void run(Socket data);
}
```

O método `run(Socket data)` que é implementado em `BancoDadosImpl` é obtém os fluxos de dados do soquete recebido através dos métodos `getOutputStream()` e `getInputStream()`. Os fluxos são depois encapsulados em filtros. Veja um trecho do método `run(Socket)`:

```
public void run (Socket con) {
      String host = con.getInetAddress().getHostName();
      try {
         in = new ComandoInputStream(con.getInputStream());
         out = new ObjectOutputStream(con.getOutputStream());
      } catch (IOException e) {
         System.out.println(this + " perdeu a conexão.");
         this.close(con);
      }
(...)
```

`ComandoInputStream` é uma subclasse do filtro `DataInputStream` desenvolvida especialmente para esta aplicação.

Protocolo de comunicações

Na comunicação entre o cliente TCP/IP (`bancodados.tier2.remote.TCPClient`) e o servidor, utilizamos um protocolo que consiste de uma série de comandos. Esses comandos são enviados pelo cliente como conjuntos de caracteres possivelmente seguidos de Strings e tipos primitivos. Após seu envio, o cliente sempre espera um objeto como resposta.

Para tornar mais simples a decodificação dos comandos enviados pelo cliente no servidor, decidimos encapsular cada comando do cliente em um objeto. O objeto que contém o comando implementa uma interface `Comando` (pacote `bancodados.tier3.tcpip`), que possui um único método:

```
public interface Comando {
   public java.lang.Object processa(BancoDados bd);
}
```

Usando esta interface, o servidor pode receber um comando do cliente e simplesmente chamar seu método `processa()`, sem precisar saber o que o comando deverá fazer. Como todo `Comando` retorna um `Object`, o servidor simplesmente retorna este object para o cliente através do `ObjectOutputStream` conectado ao seu socket.

Cada comando então deve ser implementado em uma classe própria e fornecer um procedimento para o método `processa()`. Os dados enviados pelo cliente serão formatados e convertidos em um objeto Comando correspondente antes de serem recebidos pelo servidor. Implementamos todos os comandos como classes internas da interface `BDProtocol`. Abaixo, a implementação de dois dos oito dos comandos: `length` e `setRegistro`, implementados nas classes `BDProtocol.LengthCmd` e `BDProtocol.SetRegCmd`, respectivamente:

```
public interface BDProtocol {

   /** Comando LENGTH */
   public static class LengthCmd implements Comando {

      private int numRegs;

      public Object processa(BancoDados bd) {
         numRegs =  bd.length();
         return new Integer(numRegs);
      }
   }
```

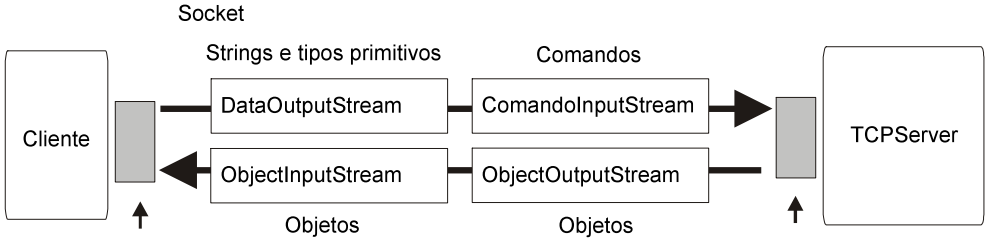


```
(...)
   /** Comando SETREGISTRO */
   public static class SetRegCmd implements Comando {

      private int numero;
      private String texto, autor;

      public SetRegCmd(int num, String txt, String aut) {
         texto = txt;
         autor = aut;
         numero = num;
      }

      public Object processa(BancoDados bd) {
         try {
            return new Boolean(bd.setRegistro(numero, texto, autor));
         }  catch (RegistroInexistente e) {
            return e.toString();  // deve ser verificado no cliente.
         }
      }
   }
(...)
```

Figura 6-6

Para utilizar os comandos acima, estendemos `java.io.DataInputStream` e acrescentamos métodos para ler seqüências de caracteres, separar palavras, identificar parâmetros e criar um objeto `Comando` correspondente. Chamamos esta classe de `ComandoInputStream` (figura 6-6).

Usando `ComandoInputStream` no servidor, recebemos comandos completos, cuja execução pode ser realizada simplesmente chamando seu método processa.



A figura 6-7 ilustra a organização dos fluxos de dados entre cliente e servidor utilizando `ComandoInputStream`.

A classe `ComandoInputStream` é construída sobre a estrutura de `DataInputStream`. Para que todos os métodos de `DataInputStream` possam ser usados como locais, é preciso que o construtor da superclasse seja chamado com o fluxo de dados passado na construção do `ComandoInputStream`:

```
public ComandoInputStream (InputStream in) {
      super(in);
}
```

A classe contém alguns métodos internos e um método público `readComando()`, listado parcialmente abaixo, que é utilizado pelo servidor para ler o próximo comando do fluxo de dados

```
   public Comando readComando() throws IOException {
      Comando cmd;
      String cmdString = readWord(); // método definido em ComandoInputStream

      if (cmdString.toLowerCase().equals("length")) {
         cmd = new BDProtocol.LengthCmd();

      } else if (cmdString.toLowerCase().equals("getregistro")) {
         cmd = new BDProtocol.GetRegCmd(readInt());  // le um inteiro

      } else if (cmdString.toLowerCase().equals("addregistro")) {
         cmd = new BDProtocol.AddRegCmd(readUTF(), readUTF());

      } else if (cmdString.toLowerCase().equals("setregistro")) {
         cmd = new BDProtocol.SetRegCmd(readInt(), readUTF(), readUTF());
(...)
      } else {
         System.out.println("Comando " + cmdString + " desconhecido!");
         return null;
      }
      return cmd;
   }
```

A execução dos comandos ocorre na segunda parte do método `run(Socket)` de `BancoDadosImpl` que fica permanentemente esperando por comandos enquanto a conexão do cliente durar. O método `readComando()` bloqueia o *thread* até que um comando esteja disponível. Quando o comando é obtido, seu método `processa()` é chamado

```
(...) // continuação do método run(Socket con)
```



```
        Comando cmd = null;
        while (cmd == null) {
           try {
              cmd = in.readComando(); // in é ComandoInputStream
              if (cmd != null) {
                    Object resultado = cmd.processa(this);
                    out.writeObject(resultado); // devolve p/ cliente
                    cmd = null;   // p/ processar proximo comando
                 }
              }
           } catch (IOException e) {
              System.out.println(this + " perdeu a conexão.");
              this.close(con);
              break;
           }
        }
(...)
```

`BancoDadosImpl` ignora a fonte de dados onde opera. Ele processa os comandos e os repassa para um `BancoDados` que pode ser qualquer implementador dessa interface.

Interface gráfica (ServerFrame)

O servidor precisa ficar executando na máquina servidora a espera de clientes. Antes de executar, precisa saber de onde vêm os dados que irá servir. Criamos então uma interface gráfica de propósito geral em `bancodados.user`, chamada `DadosServerFrame`. Ela possui a infraestrutura básica para qualquer servidor e permite que o cliente escolha um arquivo ou uma fonte de dados JDBC que o servidor irá servir. A classe `TCPDadosServerUI`, estende a classe `DadosServerFrame` para oferecer uma interface gráfica ao servidor TCP/IP.

```
public class TCPDadosServerUI extends DadosServerFrame {

   public final static int PORT = 1729;
   private BancoDados client;    // fonte de dados
   private BancoDadosImpl bdtcp;

   protected boolean init(BancoDados client) {
      if (client == null) return false;
      try {
         // cria objeto
         bdtcp = new BancoDadosImpl(client);
         // inicia servidor na porta PORT (este método é de
         // TCPServer e inicia um ServerSocket na porta PORT)
```



```
            bdtcp.startServer(PORT);
            System.out.println("Servidor no ar em: \'"+ PORT +"\'");
            return true;
        } catch (IOException ex) { /* (...) */ }
    }
```

O método `init()` é chamado quando o usuário inicia o servidor.

JAVA 7

Bancos de dados e objetos remotos

ESTE MÓDULO APRESENTA A API JDBC, que permite o acesso a bancos de dados relacionais. Apresenta também a API CORBA e RMI, que oferecem acesso a objetos remotos e à chamada de métodos remotos.

Tópicos abordados neste módulo

- O pacote `java.sql` (JDBC)
- Visão geral de ODBC e SQL
- Acesso a bancos de dados usando Java
- Os pacotes de objetos remotos `java.rmi`, `javax.rmi` e `org.omg.CORBA`
- Uso de objetos remotos e invocação de métodos
- *RMI – Remote Method Invocation*
- *CORBA – Common Object Request Broker Architecture*

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- construir uma aplicação ou applet Java que realize acesso a bancos de dados relacionais
- construir uma camada intermediária usando CORBA ou RMI
- registrar objetos em um servidor para acesso remoto (CORBA ou RMI)
- obter referências para objetos remotos
- chamar métodos em objetos remotos

7.1. JDBC

Java Database Connectivity – JDBC, é uma interface baseada em Java para acesso a bancos de dados através de SQL. Oferece uma interface uniforme para

bancos de dados de fabricantes diferentes, permitindo que sejam manipulados de uma forma consistente. O suporte a JDBC é proporcionado por uma API Java padrão `java.sql` e faz parte da distribuição Java. Usando JDBC, pode-se obter acesso direto a bancos de dados através de applets e outras aplicações Java.

JDBC é uma interface de nível de código. Consiste de um conjunto de classes e interfaces que permitem embutir código SQL como argumentos na invocação de seus métodos. Por oferecer uma interface uniforme, independente de fabricante de banco de dados, é possível construir uma aplicação Java para acesso a qualquer banco de dados SQL. A aplicação poderá ser usada com qualquer banco de dados que possua um driver JDBC: *Sybase*, *Oracle*, *Informix*, ou qualquer outro que ainda não inventado, desde que implemente um driver JDBC.

Esta seção apresentará uma introdução a JDBC e as principais classes e interfaces do pacote `java.sql`. Na seção seguinte, desenvolveremos uma aplicação de banco de dados, semelhante à do capítulo anterior (acesso a dados em arquivo de texto) mas, desta vez, oferecendo acesso a um banco de dados relacional.

Um banco de dados relacional pode ser definido de maneira simples como um conjunto de tabelas (com linhas e colunas) onde as linhas de uma tabela podem ser relacionadas a linhas de outra tabela. Este tipo de organização permite a criação de modelos de dados compostos de várias tabelas. Umas contendo informação, outras contendo referências que definem relações entre as tabelas de informação. O acesso às informações armazenadas em bancos de dados relacionais é em geral bem mais eficiente que o acesso a dados organizados seqüencialmente ou em estruturas de árvore.

SQL

Para utilizar um banco de dados relacional, é preciso ter um conjunto de instruções para recuperar, atualizar e armazenar dados. A maior parte dos bancos de dados relacionais suportam uma linguagem padrão chamada SQL – *Structured Query Language*. O conjunto de instruções SQL foi padronizado em 1992 para que as mesmas instruções pudessem ser usadas por bancos de dados diferentes. Mas vários fabricantes possuem extensões e certas operações mais específicas possuem sintaxes diferentes em produtos de diferentes fabricantes.

Poucos sistemas implementam totalmente o SQL92. Existem vários níveis que foram definidos durante a padronização do SQL em 1992. O conjunto mínimo de instruções é chamado de *entry-level* e é suportado por JDBC.

Nas seções seguintes, apresentaremos os seis principais comandos da linguagem SQL. Este não é um guia completo. Apresentamos apenas o suficiente para permitir a compreensão dos exemplos que mostraremos em JDBC.

CREATE, DROP

Antes de ilustrar a recuperação e atualização de dados em uma tabela, precisamos ter uma tabela. Para criar uma nova tabela em um banco de dados, usamos a instrução `CREATE TABLE`. A sintaxe básica desta instrução é:

```
CREATE TABLE nome_da_tabela
     (nome_coluna tipo_de_dados [modificadores],
     [nome_coluna tipo_de_dados [modificadores], ... ])
```

Os modificadores são opcionais e geralmente dependentes de fabricante de banco de dados. A maior parte das implementações suporta: `NOT NULL`, `PRIMARY KEY` e `UNIQUE` como modificadores:

```
CREATE TABLE anuncios (numero INT PRIMARY KEY,
                       data DATE,
                       texto CHAR(8192),
                       autor CHAR(50))";
```

A sintaxe exata do `CREATE` é dependente de banco de dados. Todos suportam a sintaxe principal (`CREATE TABLE`). As incompatibilidades surgem no suporte a tipos de dados e modificadores. A instrução acima funciona com o driver ODBC do *Microsoft Access*, via ponte ODBC-JDBC. Não funciona, no entanto com o driver JDBC do banco de dados *mSQL*, já que o *mSQL* não suporta o tipo `DATE`. Também não funciona com o driver ODBC para arquivos de texto, da *Microsoft* que não suporta o modificador `PRIMARY KEY` nem campos com mais de 255 caracteres.

Para remover uma tabela do banco de dados, pode-se usar a instrução `DROP`. A sintaxe é simples:

```
DROP TABLE nome_da_tabela
```

INSERT, UPDATE, DELETE

Depois que uma tabela é criada, dados podem ser inseridos usando a instrução `INSERT`. É preciso respeitar os tipos de dados definidos para cada coluna de

acordo com a estrutura definida previamente para cada tabela. A sintaxe básica do INSERT é:

```
INSERT INTO nome_da_tabela (nome_da_coluna, ..., nome_da_coluna)
       VALUES (valor, ..., valor)
```

No lugar de valor, pode ser passado toda uma expressão SQL que resulte em um valor. Um exemplo do uso de INSERT, na tabela criada na seção anterior seria:

```
INSERT INTO anuncios
       VALUES (156, '13/10/1998', 'Novo anuncio!', 'Fulano');
```

O apóstrofe (') é usado para representar strings.

UPDATE permite que dados previamente inseridos sejam modificados. Sua sintaxe básica é:

```
UPDATE nome_da_tabela
       SET nome_da_coluna = valor,
           ...,
           nome_da_coluna = valor
       WHERE expressao_condicional
```

No lugar de valor, pode ser passada toda uma expressão SQL que resulte em um valor ou a palavra reservada NULL. Eis um exemplo do uso de UPDATE:

```
UPDATE anuncios
       SET texto = 'Em branco!',
           autor = ''
       WHERE codigo > 150
```

Para remover registros (linhas da tabela), usa-se DELETE:

```
DELETE FROM nome_da_tabela
       WHERE expressao_condicional
```

Por exemplo, a instrução:

```
DELETE FROM anuncios
       WHERE texto LIKE '%Para o Lixo%'
```

apaga todos os registros cujo texto contém 'Para o Lixo'.

SELECT

O comando SQL mais freqüentemente utilizado é SELECT. Com ele é possível selecionar linhas (registros) de um banco de dados de acordo com uma determinada condição. A sintaxe básica de SELECT é:


```
SELECT nome_da_coluna, ..., nome_da_coluna
       FROM nome_da_tabela
       WHERE expressão_condicional
```

A lista de colunas a serem selecionadas pode ser substituída por "*" se todas as colunas serão selecionadas. A sintaxe mostrara acima é básica. Para selecionar todos os números e textos de registros escritos pelo autor Mefisto, pode-se fazer:

```
SELECT codigo, texto
       FROM anuncios
       WHERE autor = 'Mefisto'
```

Junções

SQL permite a realização de procedimentos bem mais complexos que os listados acima. A declaração `SELECT`, por exemplo, pode ser utilizada para realizar pesquisas em mais de uma tabela ao mesmo tempo, por exemplo:

```
SELECT anuncios.texto
       FROM anuncios, cadastros
       WHERE anuncios.autor = cadastros.autor
```

retorna o texto da tabela `anúncios` apenas quando o autor consta também da tabela `cadastros`. No caso das junções, o nome da tabela precede o nome da coluna que é separada com um ponto ".".

Pesquisa entre resultados

Pesquisas mais refinadas podem ser realizadas entre os resultados de uma pesquisa anterior colocando em cascata várias instruções `SELECT`:

```
SELECT texto
    FROM anuncios
    WHERE autor IN
         (SELECT anuncios.autor
             FROM anuncios, departamentos
             WHERE anuncios.cod_autor = departamentos.cod_autor)
```

ODBC

Para criar e administrar bancos de dados relacionais precisamos ter um ambiente próprio, geralmente fornecido pelo fabricante. Para usar esses bancos de dados dentro de aplicações, precisamos de uma maneira de encapsular o SQL dentro de uma linguagem de programação, já que embora SQL seja eficiente na administração de bancos de dados, ela não possui recursos de uma linguagem de

programação de propósito geral. Usando SQL podemos ter acesso a bancos de dados de uma forma padrão dentro de programas escritos em C ou C++ através da interface ODBC.

ODBC – *Open Database Connectivity* é uma interface de baixo nível baseada na linguagem C que oferece uma interface consistente para a comunicação com um banco de dados usando SQL. Surgiu inicialmente como um padrão para computadores desktop, desenvolvido pela *Microsoft*, mas em pouco tempo tornou-se um padrão de fato da indústria. Todos os principais fabricantes de bancos de dados dispõem de drivers ODBC.

ODBC possui diversas limitações. As principais referem-se à dependência de plataforma da linguagem C, dificultando o porte de aplicações ODBC para outras plataformas. Diferentemente das aplicações *desktop*, onde praticamente domina a plataforma *Windows*, aplicações de rede e bancos de dados freqüentemente residem em máquinas bastante diferentes. A independência de plataforma nesses casos é altamente desejável. Java oferece as vantagens de ODBC juntamente com a independência de plataforma com sua interface JDBC, que apresentaremos na próxima seção.

Muitos bancos de dados já possuem drivers JDBC, porém é possível ainda encontrar bancos de dados que não os possuem, mas têm drivers ODBC. Também, devido a ubiqüidade da plataforma Windows, que contém um conjunto de drivers ODBC nativos, é interessante poder interagir com esses drivers em várias ocasiões, por exemplo, ao montar um banco de dados SQL baseado em arquivos de texto. Como a plataforma Java contém um driver JDBC para ODBC, podemos usar JDBC em praticamente qualquer banco de dados.

Arquitetura JDBC

JDBC é uma versão Java de ODBC. É uma alternativa que acrescenta a portabilidade entre plataformas ao ODBC.

Para que se possa usar JDBC na comunicação com um banco de dados, é preciso que exista um driver para o banco de dados que implemente os métodos JDBC. O driver é uma classe Java que implementa uma interface do pacote `java.sql` chamada `Driver` e um conjunto de interfaces comuns que definem métodos usados na conexão, requisições e resposta.

Para que uma aplicação se comunique com um banco de dados, precisa carregar o driver (pode ser em tempo de execução) e obter uma conexão ao mesmo através. Isto é conseguido através de um método fábrica fornecido pela classe `DriverManager`. Depois de obtida a conexão, pode-se enviar requisições de pesquisa e atualização e analisar os dados retornados usando métodos Java e passando instruções SQL como argumentos. Não é preciso conhecer detalhes do banco de dados em questão. Em uma segunda execução do programa, o programa pode carregar outro driver e utilizar os mesmos métodos para ter acesso a um banco de dados diferente.

Muitos bancos de dados não têm driver JDBC, mas têm driver ODBC. Por causa disso, um driver JDBC para bancos de dados ODBC é fornecido pela *Sun* e incluído na distribuição Java. Com essa ponte JDBC-ODBC é possível usar drivers ODBC através de drivers JDBC. Esse driver é somente um dos quatro tipos diferentes de drivers JDBC previstos pela especificação.

A figura 7-1 ilustra um diagrama em camadas da arquitetura JDBC ilustrando os diferentes tipos de drivers.

Tipos de drivers JDBC

Existem quatro tipos de drivers JDBC [SUN]:

Tipo 1 – drivers que usam uma ponte para ter acesso a um banco de dados. Este tipo de solução geralmente requer a instalação de software do lado do cliente. Um exemplo de driver do tipo 1 é a ponte JDBC-ODBC distribuída pela *Sun* na distribuição Java.

Tipo 2 – drivers que usam uma API nativa. Esses drivers contém métodos Java implementados em C ou C++. São Java na superfície e C/C++ no interior. Esta solução também requer software do lado do cliente. A tendência é que esses drivers evoluam para drivers do tipo 3

Tipo 3 – drivers que oferecem uma API de rede ao cliente para que ele possa ter acesso a uma aplicação *middleware* no servidor que traduz as requisições do cliente em uma API específica ao driver desejado. Esta solução não requer software do lado do cliente.

Tipo 4 – drivers que se comunicam diretamente com o banco de dados usando soquetes de rede. É uma solução puro Java. Não requer código do lado do

cliente. Este tipo de driver geralmente é distribuído pelo próprio fabricante do banco de dados.

A ponte JDBC-ODBC distribuída juntamente com o JDK é um driver do tipo 1. Nos exemplos apresentados neste trabalho, utilizamos esse driver apenas para acesso local através do ODBC nativo do *Windows*. É o mais ineficiente de todos pois não permite otimizações e é dependente das limitações do driver com o qual faz ponte.

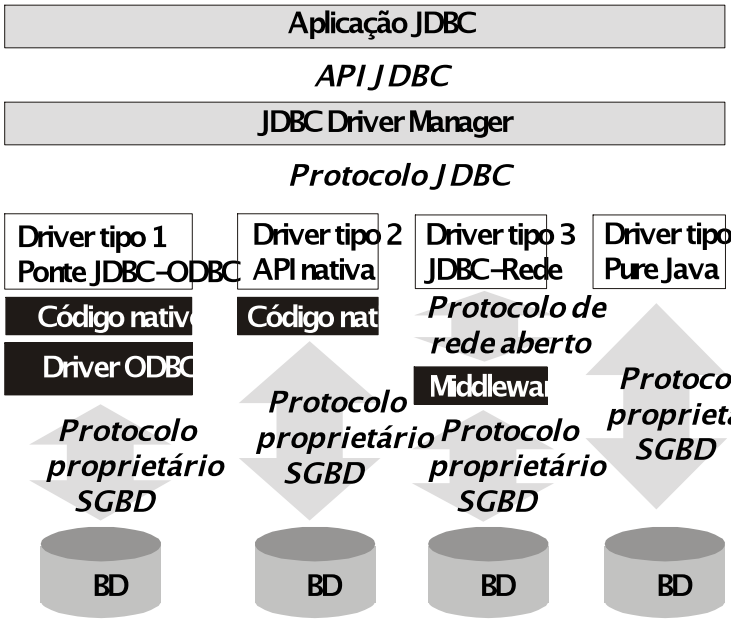


Figura 7-1: Arquitetura JDBC

Nos exemplos apresentados aqui, também utilizaremos um driver do tipo 4: o *Imaginary mSQL* driver. Com este driver, poderemos desenvolver aplicações remotas utilizando somente Java. Se no seu ambiente de trabalho houver um outro banco de dados (*Oracle*, *Informix*, *Sybase*, por exemplo), verifique se o mesmo não possui um driver JDBC próprio (tipos 2, 3 ou 4). A ponte ODBC-JDBC só deve ser usada em último caso já que carrega junto as desvantagens de falta de portabilidade e baixo desempenho.

URL JDBC

Uma aplicação JDBC pode carregar ao mesmo tempo diversos drivers. Para determinar qual driver será usado em uma conexão, uma URL é passada como argumento do método usado para obter uma conexão. Esta URL tem a sintaxe seguinte:

```
jdbc:subprotocolo:dsn
```

O `subprotocolo` é o nome do tipo de protocolo de banco de dados que está sendo usado para interpretar o SQL. É um nome dependente do fabricante. A aplicação usa o `subprotocolo` para identificar o driver a ser instanciado. O `dsn` é o nome que o `subprotocolo` utilizará para localizar um determinado servidor ou

base de dados. Pode ser o nome de uma fonte de dados do sistema local (*Data Source Name*) ou uma fonte de dados remota. Veja alguns exemplos:

```
jdbc:odbc:anuncios
jdbc:oracle:contas
jdbc:msql:clientes
jdbc:msql://alnitak.orion.org/clientes
```

A última URL, que contém um endereço Internet, não pode ser usada em drivers tipo 1. Veja na documentação do seu driver qual o nome correto para o `subprotocolo` e se o mesmo aceita acesso remoto.

Classes essenciais do pacote java.sql

Nesta seção apresentaremos as principais classes e interfaces do pacote `java.sql`, ilustradas na figura 7-2 abaixo. As classes e interfaces abaixo compõem todo o pacote `java.sql` na plataforma Java 2. As interfaces das duas primeiras colunas foram introduzidas somente nesta versão (*Java 2*). São várias interfaces para lidar com BLOBs (*Binary Large Objects*), e utilitários para melhorar a eficiência da transferência de dados.

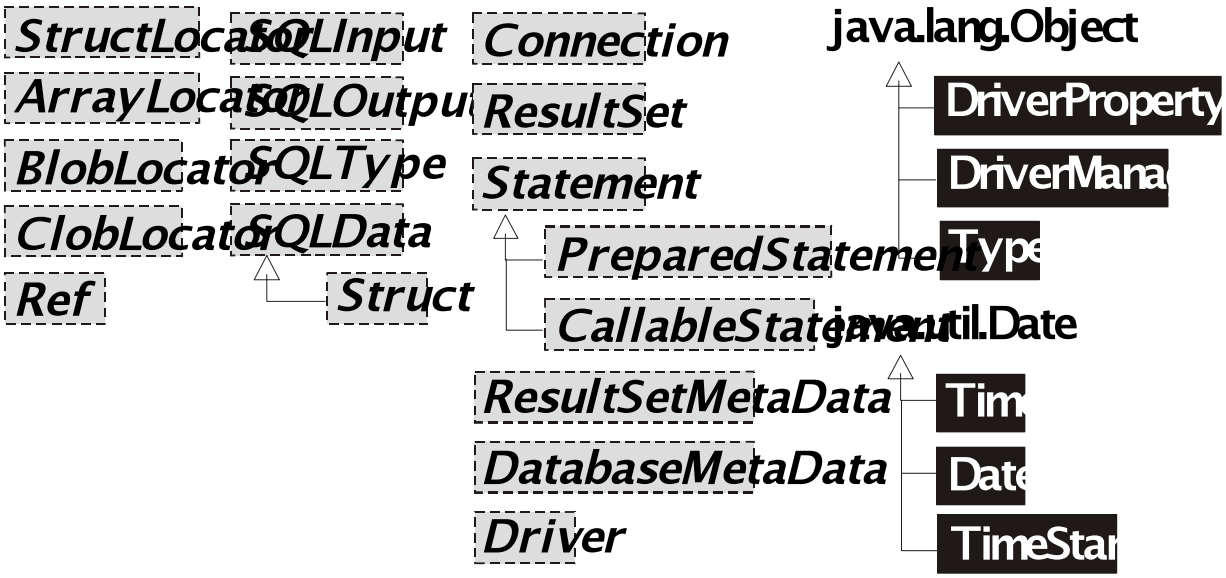


Figura 7-2: Hierarquia de classes java.sql

DriverManager e Driver

A interface `Driver` é utilizada apenas por implementações de drivers (todo driver JDBC implementa `Driver`). A classe `DriverManager` manipula objetos do tipo `Driver`. É utilizada em aplicações de acesso a banco de dados para manter

uma lista de implementações de `Driver` que podem ser usadas pela aplicação. Possui métodos para registrar novos drivers, removê-los ou listá-los. O seu método mais usado é estático e retorna um objeto do tipo `Connection`, que representa uma conexão a um banco de dados, a partir de uma URL JDBC recebida como parâmetro:

```
Connection con =
       DriverManager.getConnection("jdbc:odbc:dados","nome","senha");
```

Connection, ResultSet e Statement

As interfaces `Connection`, `Statement` e `ResultSet` contém métodos que são implementados em todos os drivers JDBC. `Connection` representa uma conexão, que é retornada pelo `DriverManager` na forma de um objeto. `Statement` oferece meios de passar instruções SQL para o sistema de bancos de dados. `ResultSet` lida com os dados recebidos.

Obtendo-se um objeto `Connection`, invoca-se sobre ele o método `createStatement()` para obter um objeto do tipo `Statement`:

```
Statement stmt = con.createStatement()
```

que poderá usar métodos como `execute()`, `executeQuery()`, `executeBatch()` e `executeUpdate()` para enviar instruções SQL ao BD. Existem duas subinterfaces de `Statement` que suportam procedimentos preparados previamente ou e armazenados no servidor: `PreparedStatement` e `CallableStatement`. Eles podem ser obtidos usando:

```
PreparedStatement pstmt = con.prepareStatement();
CallableStatement cstmt = con.prepareCall();
```

Depois de obtido um objeto `Statement`, instruções SQL podem ser enviadas para o servidor que as processará a medida em que as receber.

```
stmt.execute("CREATE TABLE dinossauros "
          + "(codigo INT PRIMARY KEY, genero CHAR(20), especie CHAR(20))");

int linhasModificadas =
    stmt.executeUpdate("INSERT INTO dinossauros "
                     + "(codigo, genero, especie) VALUES "
                     + "(499, 'Fernandosaurus', 'brasiliensis'");

ResultSet cursor =
    stmt.executeQuery("SELECT genero, especie FROM dinossauros "
                    + "WHERE codigo = 355");
```


Em vez de processar as instruções uma por uma, pode ser desejável ou necessário montar várias requisições e processá-las de uma vez. Isto pode ser obtido através do controle da lógica de transações proporcionado pela interface `Connection` através dos métodos `commit()`, `rollback()` e `setAutoCommit(boolean autoCommit)`. Por *default*, as informações são processadas a medida em que são recebidas. Isto pode ser mudado fazendo:

```
con.setAutoCommit(false).
```

Agora várias instruções podem ser acumuladas. Elas só serão processadas quando houver uma instrução `COMMIT` no banco de dados, implementada em Java da forma:

```
con.commit();
```

Se houver algum erro e todo o processo necessitar ser cancelado, pode-se emitir um `ROLLBACK` usando:

```
con.rollback();
```

O método `executeQuery()`, da interface `Statement`, retorna um objeto `ResultSet`. Este objeto implementa um ponteiro para as linhas de uma tabela. Através dele, pode-se navegar pelas linhas da tabela (registros) e recuperar as informações armazenadas nas colunas (campos).

```
ResultSet rs =
      stmt.executeQuery("SELECT Numero, Texto, Data FROM Anuncios");
while (rs.next()) {
    int x = getInt("Numero");
    String s = getString("Texto");
    java.sql.Date d = getDate("Data");
    // faça algo com os valores x, s e d obtidos...
}
```

Os métodos de navegação são `next()`, `previous()`, `absolute(int` *`linha`*`)`, `first()` e `last()`. Existem ainda diversos métodos `getXXX()` e `updateXXX()` para recuperar ou atualizar campos individuais. Os tipos retornados através desses métodos são convertidos de tipos SQL para tipos Java de acordo com a tabela 7-1.

Tabela 7-1

Método ResultSet	**Tipo de dados SQL92**
`getInt()`	`INTEGER`
`getLong()`	`BIG INT`
`getFloat()`	`REAL`

Método ResultSet	Tipo de dados SQL92
`getDouble()`	`FLOAT`
`getBignum()`	`DECIMAL`
`getBoolean()`	`BIT`
`getString()`	`CHAR, VARCHAR`
`getDate()`	`DATE`
`getTime()`	`TIME`
`getTimestamp()`	`TIME STAMP`
`getObject()`	Qualquer tipo

Após o uso, os objetos `Connection`, `Statement` e `ResultSet` devem ser fechados. Isto pode ser feito com o método `close()`:

```
con.close();
stmt.close();
rs.close();
```

DatabaseMetaData

Às vezes é preciso saber informações sobre os dados como: quantas colunas e quantas linhas existem na tabela, qual o nome das colunas, etc. Essas informações podem ser obtidas usando as classes `DatabaseMetaData` e `ResultSetMetaData`, que permitem que se obtenha informações sobre todo o bando de dados ou sobre o conjunto de dados devolvido por um `ResultSet`, respectivamente.

Para obter um objeto de metadados sobre o banco de dados, usa-se o método `getMetaData()` de `Connection`:

```
Connection con;
(...)
DatabaseMetaData dbdata = con.getMetaData();
```

Com este objeto, pode-se obter diversas informações sobre o banco de dados, como nome das tabelas, versão e nome do banco de dados, etc.:

```
String nomeDoProduto = dbdata.getDatabaseProductName();
```

Quando um `ResultSet` é retornado, podemos não saber o número e nomes das colunas. Usando um `ResultSetMetaData`, obtemos essas informações:

```
ResultSet rs;
(...)
ResultSetMetaData meta = rs.getMetaData();
int colunas = meta.getColumnCount();
String[] nomesColunas = new String[colunas];
for (int i = 0; i < colunas; i++) {
    nomesColunas[i] = meta.getColumnName(i);
```



```
}
```

Exercícios

1. Construa uma aplicação Java simples que permita que o usuário envie comandos SQL para um banco de dados e tenha os resultados listados na tela.
2. Crie uma aplicação com o mesmo objetivo que a aplicação do exercício 1, mas, desta vez, faça com que os dados sejam exibidos dentro de um `TextArea` (aplicação gráfica).
3. Crie uma interface completa para o banco de dados anuncios.mdb (disponível em disco) com três `TextFields` e uma `TextArea` para incluir, respectivamente, os campos `número`, `data`, `autor` e `texto` da tabela `anuncios`. O primeiro registro deverá ser mostrado. Implemente quatro botões de navegação. Dois para avançar e voltar um registro e dois para voltar ao início e ao fim do banco de informações.
4. Usando qualquer banco de dados disponível (que tenha driver JDBC ou ODBC) implemente uma tabela `Banco`, com código e nome de correntistas, uma tabela `Correntistas`, com código e saldo de vários correntistas. Crie agora uma interface JDBC (gráfica, se possível) que permita visualizar os dados e realizar operações bancárias (transferências, saques, depósitos, extratos e balanço).

7.2. Construção de uma aplicação JDBC

Nesta seção, apresentamos uma aplicação JDBC usando o mesmo banco de dados do capítulo anterior, desta vez organizado em um sistema de BD relacional. Para reaproveitar toda a interface do usuário e as classes que representam os conceitos fundamentais do programa, criamos uma classe `bancodados.tier2.local.BancoDadosJDBC`, que implementa a interface `bancodados.BancoDados` (veja módulo *Java 10*). Como a interface do usuário usa a interface `BancoDados`, podemos utilizar a classe `BancoDadosJDBC` preservando a mesma interface do usuário que utilizamos no módulo anterior.

Criação das tabelas

Os dados utilizados por esta aplicação serão do mesmo tipo que aqueles manipulados pela aplicação do módulo anterior. Sendo assim, teremos apenas uma tabela no banco de dados com a seguinte estrutura:

Tabela 7-2

Coluna	Tipo de dados das linhas	Informações armazenadas	Observações
`codigo`	`int`	número do anúncio	`integer` chave primária
`data`	`String`	data de postagem do anúncio	`char(24)`
`texto`	`String`	texto do anúncio	`char(8192)`
`autor`	`String`	autor do anúncio	`char(50)`

Utilizamos os tipos de dados mais comuns para garantir a compatibilidade com uma quantidade maior de bancos de dados. Poderíamos ter usado o tipo SQL `DATE`, por exemplo, no lugar de `CHAR (long)`, mas perderíamos compatibilidade com o *mSQL* que não suporta o tipo.

Antes de construir a tabela, é preciso criar uma fonte de dados ODBC e vinculá-las a uma base de dados previamente criada. Veja no apêndice A como criar bases de dados ODBC no *Windows* vinculadas a arquivos *Access* (`.mdb`) e ao sistema de arquivos (texto). Para usar uma base de dados *mSQL*, é preciso criá-la usando as ferramentas disponíveis na aplicação. Ela será acessada diretamente.

Construída a base de dados, podemos executar o programa `bancodados.util.CriaTabelas` ou `CriaTabelas2` para os bancos de dados ODBC baseados em arquivo de texto (as aplicações estão disponíveis no subdiretório `jad/apps/bancodados/util`). Qualquer uma das aplicações requer como parâmetros:

- uma URL JDBC do tipo `jdbc:odbc:`*<nome da fonte de dados>* para acesso ODBC ou `jdbc:msql://localhost/`*<nome da base de dados>* para acesso via *mSQL*.
- nome do usuário
- a senha do usuário

Por exemplo, para criar as tabelas em uma fonte de dados ODBC local chamada `classif`:

```
java bancodados.util.CriaTabelas jdbc:odbc:classif scott tiger
```


Se as tabelas forem criadas com sucesso, será impressa uma mensagem:

```
Tabelas criadas com sucesso!"
```

A listagem abaixo esclarece as partes mais importantes do código-fonte. Veja o código completo com comentários em `jad/apps/bancodados/util`.

O método `main()`, que inicia a aplicação, primeiro faz uma verificação para checar se os parâmetros requeridos foram passados na linha de comando. Caso positivo, inicializa o objeto passando os argumentos de linha de comando para o construtor. Quando o construtor terminar, será chamado o método `criaTabela()`.

```
    public static void main(String[] args) {
       if (args.length < 1) {
          System.out.println
            ("Sintaxe correta: java CriaTabelas url_jdbc [nome] [senha]");
       (...)
          System.exit(1);
       }
       String url = args[0];
       String nome = "";
       String senha = "";

       if (args.length >= 2) {
          nome = args[1];
       }
       if (args.length >= 3) {
          senha = args[2];
       }

       CriaTabelas ct = new CriaTabelas(url, nome, senha);
       ct.criaTabela();
    }
}
```

A aplicação localiza os drivers disponíveis através da informação armazenada em um arquivo `drivers.jdbc` armazenado no diretório de trabalho da aplicação (`jad/apps`). Define ainda variáveis de instância para representar a conexão e o contexto de envio de declarações SQL. O arquivo `drivers.jdbc` permite que novos bancos de dados sejam utilizados e que seus drivers JDBC (classes Java) sejam carregados sem que seja necessária a recompilação do programa. O arquivo relaciona nomes de classes e subprotocolos. Eis a listagem da classe `CriaTabelas.java`:


```
package bancodados.util;
 (...)
public class CriaTabelas {

   // arquivo com drivers disponíveis
   // localizado no diretorio onde roda a aplicacao
   private static final String DRIVERS = "drivers.jdbc";

   private Connection con;              // conexao
   private Statement stmt;;             // declaracao SQL
   private Vector numeros;              // vetor com numeros de registro
```

O construtor da aplicação tem duas partes. Inicialmente, tenta carregar o arquivo onde estão listados os drivers disponíveis. Isto é feito através de um método local chamado `loadDrivers()`. Se consegue, o método retorna uma matriz com o subprotocolo e nome completo da classe Java que implementa o driver. Se falha, usa o driver ODBC (*default*).

```
   public CriaTabelas(String url, String nome, String senha) {

      // nomes de drivers disponíveis: default ponte JDBC-ODBC da Sun
      String driv = "odbc";
      String[][] drivers = {{"odbc", "sun.jdbc.odbc.JdbcOdbcDriver"}};
      boolean temDriver = false;

      try {
         try {
            // método loadDriverNames é local (veja código fonte).
            // Ele lê arquivo de drivers e devolve matriz com
            // [subprotocolo] [nome do driver]
            String[][] listaDrivers = loadDriverNames(DRIVERS);

            // Leu arquivo...  redefine drivers
            drivers = listaDrivers;
            listaDrivers = null;    // libera o objeto

            // agora faça validação da URL recebida pelo cliente
            StringTokenizer st = new StringTokenizer(url, ":");
            String prot = st.nextToken();   // obtem primeiro token
            if (!prot.equals("jdbc"))
               throw new java.net.MalformedURLException
                         ("A URL tem que ser do tipo 'jdbc:'!");

            driv = st.nextToken();  // obtem segundo token

         } catch (FileNotFoundException e) {
```



```
            // se nao existe arquivo de configuracao de drivers,
            // continue com o driver default
            (...)
            temDriver = true;
            // continua...
         }
```

A segunda parte do construtor prossegue verificando se o protocolo da URL passada na linha de comando coincide com o subprotocolo de algum driver disponível. Se coincide, o driver é carregado, caso contrário, uma exceção é provocada e o programa é terminado.

```
         for (int i = 0; i < drivers.length; i++) {
            // verifica se subprotocolo coincide com driver disponivel
            if (drivers[i][0].equals(driv)) {
               Class.forName(drivers[i][1]);    // carrega driver
               temDriver = true; // pelo menos um driver foi carregado
            }
         }

         if (temDriver == false) {
            throw new UnsupportedOperationException
                ("Não há drivers disponíveis para " + url + "!");
         }
```

No final, tendo-se carregado o driver, a URL, nome e senha passados como argumentos de linha de comando são utilizados para se obter uma conexão ao banco de dados (através do objeto `Connection`). Logo em seguida, é obtido um objeto `Statement`. A partir de agora, métodos que executam SQL podem ser chamados.

```
            con = DriverManager.getConnection(url, nome, senha);  // obtem conexao
            stmt = con.createStatement();                         // cria statement

         } catch (Exception e) {
            System.out.println(e);
            System.exit(0);
         }
      }
```

Logo que o construtor termina seu trabalho, o método `criaTabela()` é chamado. Este método simplesmente define um procedimento SQL e o passa como argumento do método execute invocado sobre o objeto `stmt`. Se não ocor-

rer exceção, as tabelas serão criadas e a aplicação imprimirá uma mensagem indicando sucesso.

```
public void criaTabela() {
   String create = "CREATE TABLE anuncios (numero int primary key, " +
                                          " data char(24)," +
                                          " texto char(8192)," +
                                          " autor char(50))";

   try {
      stmt.execute (create);
      System.out.println("Tabelas criadas com sucesso!");
   } catch (SQLException e) {
      e.printStackTrace();
   }
}
```

Inicialização

A figura 7-3 ilustra o diagrama da aplicação. Somente as interfaces que estão na parte superior do desenho (`BancoDados` e `BancoDadosJDBC`) são acessíveis e alteráveis dentro da nossa aplicação. O código que usa `java.sql` está somente na classe `BancoDadosJDBC` que usa `DriverManager` para obter um driver (objeto do tipo `Driver`), através do qual interage com o banco de dados ODBC.

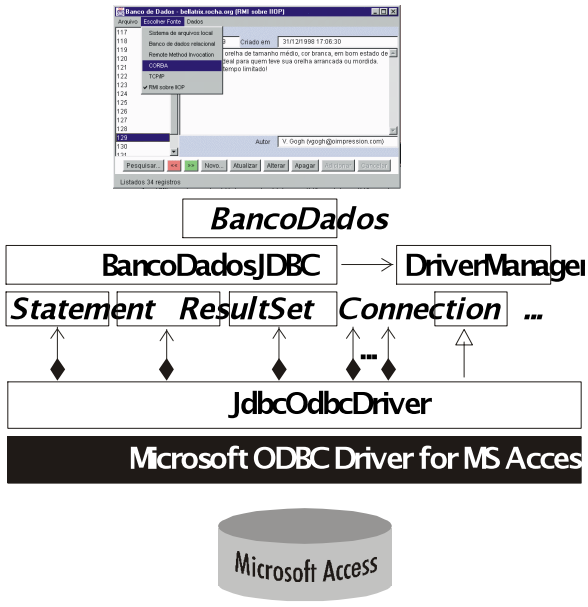


Figura 7-3: diagrama da aplicação mostrando JDBC interfa-

Na classe `BancoDadosJDBC` carregamos a classe e criamos uma instância de um driver de acordo com a URL passada pelo usuário, da mesma forma como fizemos na classe `CriaTabelas`.

O construtor da classe `BancoDadosJDBC` é bem parecido com o construtor da aplicação `CriaTabelas`. Contém as seguintes instruções:

```
public BancoDadosJDBC(String url, String nome, String senha) throws IOException

      // Define formato para todas as datas (dd/mm/aaaa hh:mm:ss)
      df = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM);
(...)
      try {
         try {
```



```
            String dir = System.getProperty("app.home");
            String[][] listaDrivers = loadDriverNames(dir +
                                      File.separator + DRIVERS);
            // Leu arquivo...  redefine drivers
            drivers = listaDrivers;
(...)
         for (int i = 0; i < drivers.length; i++) {
            if (drivers[i][0].equals(driv)) {
               Class.forName(drivers[i][1]);    // carrega driver
               temDriver = true;
            }
         }
 (...)
         con = DriverManager.getConnection(url, nome, senha);  // obtem conexao
         stmt = con.createStatement();                         // cria statement

      } catch (Exception e) { // (...)
      }
   }
```

Os objetos `con` e `stmt` são do tipo `Connection` e `Statement`, respectivamente (veja o código). Foram declarados como variáveis de instância do objeto `BancoDadosJDBC`.

A instrução `Class.forName()` carrega a classe do driver pelo nome. O `DriverManager` recebe uma URL JDBC, informada pelo cliente, localiza um driver, cria uma instância dele e obtém um objeto do tipo `Connection`. Depois, usa o método `createStatement()` para obter um objeto `Statement` que será usado na implementação dos métodos de `bancodados.BancoDados`.

Acesso ao banco de dados

A referência `stmt` é usada em todos os métodos. Alguns retornam uma tabela com valores, outros não. O método `executeUpdate` pode ser usado para inserir registros na implementação de `addRegistro()`, que acrescenta um novo registro no banco de dados:

```
public synchronized void addRegistro(String texto, String autor) {
      ResultSet rs;
      int numero = getProximoNumeroLivre();
      java.util.Date data = new java.util.Date();
      String quando = df.format(data);
      String insert = "INSERT INTO anuncios VALUES (" + numero + ", '"
                                                     + quando + "', '"
```



```
                                                    + texto +  "', '"
                                                    + autor +  "')";
      try {
         stmt.executeUpdate(insert);
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```

O método `getRegistros` seleciona todos os números de registro do servidor e os coloca em um `Vector`. Veja a implementação JDBC e compare com a implementação mostrada no ultimo capítulo. Neste caso, espera-se o retorno de uma tabela de dados como resposta, usamos o método `executeQuery()`, que retorna um `ResultSet` cujo ponteiro pode ser adiantado com o seu método `next()`. Para cada posição (que corresponde a uma linha da tabela `ResultSet`), usamos os métodos `getXXX()` apropriados para ler inteiros, strings e datas.

```
public Registro[] getRegistros() {
     ResultSet rs;
     Vector regsVec = new Vector();
     String query = "SELECT numero, data, texto, autor " +
                    "FROM anuncios ";
     try {
        rs = stmt.executeQuery(query);
        while (rs.next()) {
           int numero = rs.getInt("numero");
           String texto = rs.getString("texto");
           String dataStr = rs.getString("data");
           java.util.Date data = df.parse(dataStr);
           // java.util.Date data = rs.getDate("data");
           String autor = rs.getString("autor");
           regsVec.addElement(new Registro(numero, texto, data, autor));
        }
     } catch (SQLException e) { // (...)
     } catch (java.text.ParseException e) { // (...)
     }
     Registro[] regs = new Registro[regsVec.size()];
     regsVec.copyInto(regs);
     return regs;
  }
```

A remoção do registro é implementado de forma mais simples ainda. É só encapsular uma instrução SQL `DELETE`:

```
public synchronized boolean removeRegistro(int numero)
```



```
        throws RegistroInexistente {

        ResultSet rs;
        String delete = "DELETE FROM anuncios WHERE numero = " + numero;

        try {
           stmt.executeUpdate(delete);
           return true;
        } catch (SQLException e) {
           e.printStackTrace();
           return false;
        }
     }
```

Interface do usuário

A única alteração necessária na interface do usuário para o acesso ao banco de dados JDBC, é o fornecimento de uma interface de entrada de dados para que o cliente possa escolher a fonte de dados ao qual quer conectar-se, seu nome e senha de acesso. Na aplicação gráfica, utilizamos uma janela de diálogo para coletar esses dados (veja figura 10-2c). Na aplicação de texto, utilizamos a linha de comando:

```
---- Cliente de Banco de Dados ----
Digite o numero correspondente a forma de conexao:
1) Sistema de arquivos local
2) Banco de dados relacional
3) Remote Method Invocation
4) CORBA
5) TCP/IP
6) RMI sobre IIOP
10) Sair do programa
--> 2
Informe a URL JDBC para conexao: jdbc:odbc:txtdados1
Informe o id do usuario para acesso [nenhuma]: helder
Informe a senha para acesso [nenhuma]: kz9919
```

e passamos as informações recebidas na construção do objeto:

```
(...)
case 2:     // JDBC
    System.out.print("Informe a URL JDBC para conexao: ");
    System.out.flush();
    String url = br.readLine();
    System.out.print("Informe o id do usuario para acesso [nenhuma]: ");
    System.out.flush();
    String nome = br.readLine();
```



```
        System.out.print("Informe a senha para acesso [nenhuma]: ");
        System.out.flush();
        String senha = br.readLine();
        System.out.print("Conectando a " + url + "...");
        System.out.flush();
        client = new BancoDadosJDBC(url, nome, senha);
        System.out.println("Conectado a " + url + "!");
        menuOpcoes();
    break;
    (...)
```

A referência `client`, no código acima, é do tipo `bancodados.BancoDados`.

7.3. CORBA

A última seção apresentou uma solução que permitia transformar uma aplicação de uso local em uma aplicação distribuída. Para atingir esse objetivo, tivemos que elaborar regras de comunicações próprias e implementar um cliente e servidor específicos, conforme o protocolo estabelecido.

Utilizando uma tecnologia de objetos distribuídos, podemos obter os mesmos resultados usando um protocolo padrão e desenvolvendo um código mais simples e com maiores possibilidades de reutilização. Esta seção apresenta uma breve introdução à tecnologia de objetos distribuídos CORBA e como poderíamos utilizá-la com a linguagem Java. Na seção seguinte, mostraremos como, através de um procedimento passo-a-passo, podemos implementar uma camada intermediária baseada em CORBA para a aplicação de banco de dados explorada nos módulos e seções anteriores.

Por que usar CORBA?

No módulo anterior vimos que é possível implementar aplicações distribuídas independentes de plataforma em Java usando as classes do pacote `java.net`. Para atingir nossa meta tivemos que desenvolver um protocolo e criar clientes e servidores que se utilizassem desse protocolo para estabelecer um canal de comunicação. Tivemos que definir portas, administrar *threads* individuais para cada conexão e garantir a correta abertura e fechamento dos soquetes e fluxos de dados. Usando agora a tecnologia CORBA, nossa meta é obter a mesma conectividade, mas reduzindo de forma significativa a complexidade.

CORBA oferece uma infraestrutura que permite a invocação de operações em objetos localizados em qualquer lugar da rede como se fossem locais à aplicação que os utiliza. Assim sendo, poderemos invocar métodos remotos como se fossem locais e não precisar se preocupar com protocolos, portas, localidades, *threads* ou controle de fluxos de dados.

Mas CORBA oferece mais que isto. Oferece também independência em relação à linguagem na qual foram desenvolvidos os objetos. A especificação CORBA 2.2 permite que programas distribuídos tenham objetos escritos em *C*, *C++*, *Smalltalk*, *COBOL* e *Java*, e que todos possam se comunicar entre si. Portanto, CORBA é uma ótima solução para desenvolver clientes Java para aplicações legadas, escritas em *C*, *C++*, *COBOL* ou *Smalltalk* e situadas em máquinas remotas.

Fundamentos de CORBA e IDL

CORBA – Common ORB Architecture (arquitetura comum do ORB) é uma especificação que estabelece uma forma de comunicação entre aplicações independente da plataforma onde residem e da linguagem em que foram escritas. O coração da arquitetura CORBA é o *ORB – Object Request Broker* (corretor de requisição de objetos), um "barramento" de mensagens que intercepta as requisições dos clientes CORBA e invoca operações em objetos remotos, passando parâmetros e retornando os resultados da operação ao cliente. Em suma, ORBs servem para transferir objetos de um lugar para outro. Eles ORBs cuidam da forma como os objetos são transferidos deixando os detalhes ocultos ao programador ou usuário. Um ORB, roda em uma única máquina, mas pode se conectar a todos os outros ORBs disponíveis em uma rede TCP/IP, usando o protocolo IIOP (*Internet Inter-ORB Protocol*).

A especificação CORBA é desenvolvida pelo *OMG – Object Management Group*, o maior consórcio da indústria de computadores com mais de 750 membros (1997). A OMG é uma organização sem fins lucrativos cujos objetivos são promover teoria e prática da engenharia de software orientada a objetos, oferecendo uma arquitetura comum para o desenvolvimento de aplicações independente de plataformas de hardware, sistemas operacionais e linguagens de programação. A meta da OMG visa reduzir a complexidade, custos e como conseqüência levar ao desenvolvimento de novas aplicações.

Essencial na tecnologia CORBA está a noção de transparência que ocorre de duas formas. *Transparência em relação à localidade* estabelece que deve ser tão simples invocar operações em um objeto remoto como seria fazer o mesmo se aquele objeto estivesse disponível localmente. *Transparência em relação à linguagem de programação* garante a liberdade para implementar a funcionalidade encapsulada em um objeto usando a linguagem mais adequada, seja qual for o motivo. Isto é conseguido separando a implementação do objeto da sua interface pública de acesso, que é definida usando uma linguagem de definição de interface (IDL – *Interface Definition Language*) neutra quanto à implementação.

IDL – Interface Definition Language

A linguagem OMG IDL é uma linguagem declarativa que oferece uma sintaxe e nomes genéricos para representar estruturas comuns como módulos, interfaces, métodos, tipos de dados, etc. em objetos CORBA. O objeto é completamente especificado na IDL. Para ter utilidade, deve haver um *mapeamento* entre o IDL e a linguagem de programação na qual os objetos estão implementados. Fabricantes de ORBs fornecem um *compilador IDL* cuja finalidade é gerar módulos de suporte às operações declaradas na interface na linguagem de implementação.

Um arquivo IDL declara a interface pública de um objeto. Informa o nome das operações suportadas pelo objeto, o tipo de seus parâmetros, o tipo de suas variáveis públicas e os tipos que são retornados após a invocação de um método. A listagem a seguir mostra um exemplo de OMG IDL com as palavras reservadas em negrito.

```
module BDImagens {       // módulo

   exception ErroEntradaSaida{};          // exceções
   exception FimDeArquivoPrematuro{};

   interface Registro {        // interface
      long getCodigo();
      string getNome();
      string getTipo();        // operações
   };

   interface BancoDados {
      void setRegistro(in long cod, in string nome, in string tipo)
          raises (ErroEntradaSaida);
```



```
        Registro getRegistro()
            raises (ErroEntradaSaida, FimDeArquivoPrematuro);
    };

    interface Prancheta {
        typedef sequence <octet> Color;      // definição de tipos
        typedef sequence <long>  IntArray;

        // Retorna a imagem do desenho.
        IntArray getDesenho();

        // Retorna a cor atual.
        Color getCorAtual();

        // Redefine a imagem do desenho.
        void setDesenho(in IntArray newBitmap);
    };
};
```

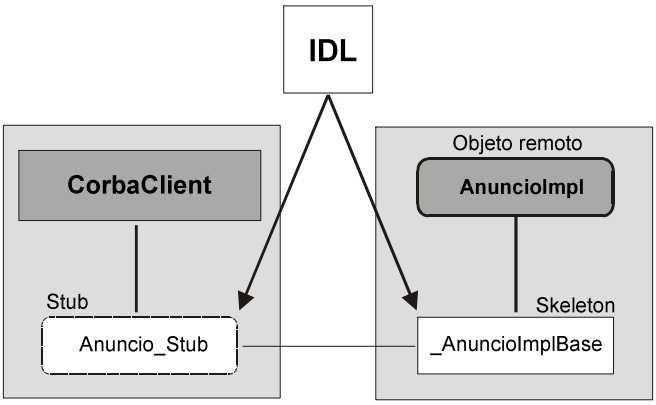


Figura 7-4 Um mesmo IDL define o stub na linguagem do cliente e um skeleton na linguagem do objeto remoto, permitindo que o cliente invoque métodos no objeto remoto mesmo estando escrito em outra linguagem.

A IDL da OMG tem sua sintaxe derivada do C++, mas contém palavras-chave diferentes e específicas. Um arquivo IDL é tudo que um programador necessita para implementar o cliente para um determinado objeto. O compilador IDL, cria os arquivos fonte que implementam a interface para o cliente e o servidor do objeto, na linguagem específica do ORB que fornece o compilador. Cada compilador obedece a um determinado mapeamento IDL-linguagem.

Para o cliente, são gerados arquivos *stub* – que contêm código que realiza a tarefa de intermediar, de forma transparente ao programador e usuário, as operações entre o cliente e o ORB. O *stub* dá ao cliente a ilusão de estar acessando diretamente o objeto remoto, transformando os tipos de dados da linguagem de programação no qual foi criado em um formato de dados interpretado pelo ORB como uma mensagem ao objeto remoto.

Do outro lado, no servidor, arquivos *skeleton* fazem o trabalho inverso, convertendo a requisição na linguagem de programação e tipos de dados locais ao servidor. Arquivos auxiliares também podem ser gerados pelo compilador de forma a tornar a implementação do servidor de objetos transparente em relação à existência do *skeleton*.

Um mesmo IDL pode ser usado na geração de *stubs* e *skeletons* de linguagens diferentes (figura 7-4), permitindo a comunicação de aplicações implementadas em linguagens distintas. Por exemplo, pode-se gerar um *stub* de cliente Java para um objeto com interface definida em uma determinada IDL usando um compilador com mapeamento IDL-Java. Usando o mesmo IDL, e um compilador com mapeamento IDL-COBOL, pode-se criar um *skeleton* para um servidor de objetos remotos em COBOL, fazendo assim uma aplicação Java interagir de forma transparente com uma aplicação COBOL.

Geralmente, objetos CORBA em um ambiente distribuído são servidos por ORBs diferentes. A comunicação entre ORBs é realizada através de protocolos Inter-ORB (*IOP – Inter-ORB Protocol*). Na Internet e redes TCP/IP a conectividade é possível via IIOP – *Internet Inter ORB Protocol*.

Usando CORBA com Java

Para utilizar CORBA com Java é preciso ter um ORB que implemente um mapeamento IDL-Java. Há uma correspondência entre construções IDL e estruturas presentes na linguagem Java. A tabela abaixo mostra o mapeamento entre tipos de dados, definido na especificação CORBA 2.2. Este mapeamento é utilizado pelo compilador na geração de *stubs* e *skeletons*.

Tabela 7-3

Tipo IDL	Tipo Java
boolean	boolean
char	char
wchar	char
octet	byte
short/unsigned short	short
long / unsigned long	int
long long / unsigned long long	long
float	float
double	double

Estruturas como pacotes, classes, interfaces, objetos serializados, exceções também têm uma correspondência simples em IDL. Como Java é uma linguagem orientada a objetos, a conversão de uma interface IDL em uma interface Java é muito simples. Veja na tabela 7-4, algumas estruturas Java e seus pares em IDL:

Tabela 7-4

IDL	Java
module	package
interface	class, interface
raises	throws
exception	Exception
string	String
void	void
enum, union, struct	class
sequence <octet>,	byte[], int[], ...
sequence <long>, ...	(vetores e objetos serializados)
:	extends
const (dentro de interfaces)	static final

A listagem a seguir representa três interfaces Java e duas exceções que podem ser representadas pela interface IDL listada no início desta seção. Veja e compare as duas listagens.

```
package BDImagens;

class ErroEntradaSaida extends Exception {}
class FimDeArquivoPrematuro extends Exception {}

interface Registro {
    public int getCodigo();
    public String getNome();
    public String getTipo();
}

interface BancoDados {
    public void setRegistro(int cod, String nome, String tipo)
           throws ErroEntradaSaida;

    public Registro getRegistro()
           throws ErroEntradaSaida, FimDeArquivoPrematuro;
}

interface Prancheta {
    public int[] getDesenho();
```



```
    public byte[] getCorAtual();
    public void setDesenho(int[] newBitmap);
}
```

Já existem vários ORBs que oferecem suporte a Java. Objetos CORBA desenvolvidos para uma ORB podem não rodar em outro ORB pois poderá haver diferenças na implementação do IDL em ORBs de fabricantes diferentes. O IDL, porém, sempre pode ser usado para desenvolver um objeto compatível com o ORB disponível. Neste trabalho, utilizaremos o ORB nativo do Java 2 por ser gratuito e disponível automaticamente com a distribuição Java.

O suporte a CORBA no Java 2 está nos pacotes e subpacotes de `org.omg.CORBA`, nos pacotes `org.omg.CosNaming` e `omg.CosNaming.NamingContext-Package` (serviço de registro de nomes), e nas ferramentas `idltojava`, `idlj` (compiladores IDL-Java) e `tnameserv` (servidor de registro de nomes).

Exercícios

5. Defina uma interface IDL para enviar requisições SQL remotas para aplicação proposta no exercício 1 (ou 2). Implementa a aplicação em Java e faça os testes, preferencialmente usando máquinas diferentes.
6. Crie uma interface IDL para a aplicação bancária e implemente o acesso remoto nos serviços bancários.

7.4. Construção de aplicações distribuídas com CORBA

A melhor maneira de compreender o funcionamento de Java com CORBA é através de exemplos. O restante desta seção será dedicada à implementação de uma camada CORBA na aplicação de banco de dados vista na seção anterior.

Aplicação de banco de dados

Esta aplicação consiste do quadro de anúncios classificados, semelhante ao apresentado nos capítulos anteriores, mas com um servidor intermediário CORBA. O cliente pode, através de um cliente remoto, realizar operações como acrescentar um novo anúncio, listar os anúncios existentes, alterá-los, apagá-los, pesquisar, etc. Para isto, é preciso que ele obtenha uma referência remota do objeto que representa o quadro de avisos e invocar os seus métodos.

Interface IDL

O primeiro passo é definir a interface IDL que irá descrever os objetos que serão usados ou implementados. Criamos um arquivo `bdcorba.idl` com o seguinte conteúdo, representando as classes `Registro` e `BancoDados` da aplicação no servidor pelas interfaces IDL `Anuncio` e `QuadroAvisos`, respectivamente:

```
module bancodados {
 module util {
  module corba {

   // definição de tipo Date serializado (byte[])
   typedef sequence <octet> Date;
   typedef sequence <octet> AnuncioArray;

   // tratamento para IOException e RegistroInexistente
   exception ErroES{};
   exception NaoExiste{};

   // definição de objeto utilizado pelo quadro de avisos
   interface Anuncio {
      long getNumero();        // retorna numero do anuncio
      string getTexto();       // retorna conteudo do anuncio
      Date getData();          // retorna data postagem do anuncio
      string getAutor();       // retorna endereco do anunciante
   };

   // quadro de avisos
   interface QuadroAvisos {
      long length() raises (ErroES);
      void addRegistro(in string anuncio,
                       in string contato) raises (ErroES);
      Anuncio getRegistro(in long numero)
           raises (ErroES, NaoExiste);      // obtem registro
      boolean setRegistro(in long numero,
                          in string anuncio,
                          in string contato) raises (NaoExiste);
      boolean removeRegistro(in long numero) raises (NaoExiste);
      AnuncioArray getRegistros(in string textoProcurado);
      AnuncioArray getAllRegistros();
   };
  };
 };
};
```


Anuncio e `QuadroAvisos` são interfaces equivalentes a `Registro` e `BancoDados`. Usamos nomes diferentes para evitar conflitos (pois o IDL irá gerar classes com esses nomes) e podermos importar todo o pacote `bancodados` nas classes utilizadas.

O arquivo IDL possui quatro partes. Todas estão dentro do submódulo "`corba`" que será implementado como um pacote Java pelo compilador IDL. A primeira declaração:

```
typedef sequence <octet> Date;
```

define o tipo `Date`, como uma seqüência de bytes. Esta seqüência, em Java, poderá ser convertida em um objeto `java.util.Date` na nossa implementação. As declarações:

```
exception ErroES{};
exception NaoExiste{};
```

definem exceções provocadas pelos métodos da interface `QuadroAvisos`. As exceções definidas pelo usuário serão implementadas pelo compilador IDL como classes finais que estendem `org.omg.CORBA.UserException`.

As duas declarações seguintes

```
interface Anuncio { ... };
interface QuadroAvisos { ... };
```

definem as interfaces aos objetos que o cliente terá acesso. A definição de `Anuncio` é necessária porque o `QuadroAvisos` retorna um objeto deste tipo no método `getRegistro()`. Dentro de cada definição estão declaradas as operações (métodos) que poderão ser invocadas remotamente sobre os objetos `QuadroAvisos` e `Anuncio`.

Vejamos algumas das operações declaradas em `QuadroAvisos`:

```
boolean length() raises (ErroES);

void addRegistro(in string anuncio,
                 in string contato) raises (ErroES);

Anuncio getRegistro(in long numero) raises (ErroES, NaoExiste);
```

A semelhança com métodos Java é bastante grande. Observe que `raises` tem a mesma função que `throws`. Veja o formato dos parâmetros da operação `setRegistro`. Os parâmetros precedidos por "`in`" são semelhantes à parâmetros de métodos Java, passados pelo objeto que invoca a operação. Os parâmetros pre-

cedidos por "`out`" e "`inout`" (não utilizados aqui) não têm equivalente em Java. Esse indicador significa que o valor para o parâmetro será fornecido pelo objeto invocado. O suporte a esse tipo de operação é feito através de objetos especiais (*Holder*) criados pelo compilador IDL-Java.

Compilação IDL-Java

O segundo passo é compilar o arquivo IDL para gerar o código de suporte ao cliente (*stub*) e código de suporte aos objetos no servidor (*skeleton*). No nosso exemplo, tanto o cliente quanto o servidor serão desenvolvidos em Java, mas, nada impede que se utilize o mesmo IDL para gerar um servidor em C, C++, Smalltalk ou COBOL para a comunicação com um cliente Java e vice-versa.
O Java 2 fornece um compilador IDL chamado `idltojava` (`idltojava.exe` em *Windows*). Para compilar o `bdcorba.idl`, faça:

```
idltojava -fno-cpp bdcorba.idl
```

Com o arquivo IDL no subdiretório `jad/apps/`, o compilador terá criado o subdiretório "`bancodados/util/corba`" abaixo do diretório atual. Este subdiretório representa um pacote que contém as seguintes classes:

`DateHolder.java` `DateHelper.java`	Classes de suporte ao tipo definido `Date`
`ErroES.java`	Implementação Java da exceção `ErroES`
`ErroESHelper.java` `ErroESHolder.java`	Classes de suporte ao tipo `ErroES`
`NaoExiste.java`	Implementação Java da exceção `NaoExiste`
`NaoExisteHelper.java` `NaoExisteHolder.java`	Classes de suporte ao tipo `NaoExiste`
`_AnuncioStub.java`	*Stub* do cliente para o objeto remoto `Anuncio`
`Anuncio.java`	Interface Java isomórfica à interface IDL `Anuncio`
`AnuncioHolder.java` `AnuncioHelper.java`	Classes de suporte ao tipo `Anuncio`
`_QuadroAvisosStub.java`	*Stub* do cliente para o objeto remoto `QuadroAvisos`
`QuadroAvisos.java`	Interface Java isomórfica à interface IDL `QuadroAvisos`
`QuadroAvisosHolder.java` `QuadroAvisosHelper.java`	Classes de suporte ao tipo `QuadroAvisos`
`_AnuncioImplBase.java`	*Skeleton* do servidor. Implementa a interface `Anuncio`
`_QuadroAvisosImplBase.java`	*Skeleton* do servidor. Implementa `QuadroAvisos`

Implementação do servidor

A próxima etapa é implementar os objetos remotos utilizando como base os *skeletons* gerados na compilação do IDL. Neste modelo de implementação CORBA, precisamos estender a classe esqueleto. No nosso exemplo, devemos estender `_AnuncioImplBase` e `_QuadroAvisosImplBase`. A listagem abaixo (parcial) mostra a implementação deste último. Os outros arquivos-fonte da implementação CORBA podem ser encontrados em `jad/apps/bancodados/tier3/corba`.

```
package bancodados.tier3.corba;

import bancodados.*;
import bancodados.util.*;
import bancodados.util.corba.*;

public class QuadroAvisosImpl extends _QuadroAvisosImplBase {

   private BancoDados fonte;       // acesso a operacoes de cliente local

   public QuadroAvisosImpl(BancoDados fonte) {
      this.fonte = fonte;
   }

   public int length() throws ErroES {
      return fonte.length();
   }

   public synchronized void addRegistro(String anuncio, String contato)
      throws ErroES {
      fonte.addRegistro(anuncio, contato);
   }

   public boolean setRegistro(int numero, String anuncio, String contato)
      throws NaoExiste {
      try {
         return fonte.setRegistro(numero, anuncio, contato);
      } catch (RegistroInexistente ri) {
         throw new NaoExiste();
      }
   }

   public boolean removeRegistro(int numero) throws NaoExiste {
      try {
         return fonte.removeRegistro(numero);
      } catch (RegistroInexistente ri) {
```



```
            throw new NaoExiste();
         }
      }
(...)
}
```

Observe que a classe acima usa a classe `bancodados.BancoDados` para ter acesso, como cliente, à terceira camada da aplicação, ficando, desta forma, totalmente independente dos detalhes de implementação da terceira camada. O cliente (`fonte`) é passado como argumento na construção do objeto, através do servidor.

A próxima etapa é a implementação do servidor que exportará e oferecerá acesso aos objetos remotos. Ele terá que criar uma instância do objeto a ser distribuído (`QuadroAvisosImpl`) e registrá-lo no serviço de nomes. O servidor é uma classe executável que deverá permanecer no ar enquanto houver interesse de servir clientes remotos. É uma aplicação gráfica semelhante ao servidor TCP/IP do último capítulo, pois estende `bancodados.user.DadosServerFrame`. Usa as classes do pacote `org.omg.CosNaming` para exportar e registrar o objeto no sistema de nomes. Mostramos abaixo o construtor e o método `init()` que contém a inicialização e exportação dos objetos:

```
package bancodados.user;

import java.io.*;
import java.util.Properties;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

(...)
public class CorbaDadosServerUI extends DadosServerFrame {

   public final static String NOMESERV = "anuncios";
   private BancoDados client;
   private QuadroAvisosImpl bdc;
   private ORB orb;

   public CorbaDadosServerUI(String[] args) {
      super(frameTitle);

      //inicializacao do ORB
      Properties props = new Properties();
      props.put("org.omg.CORBA.ORBClass", "com.sun.CORBA.iiop.ORB");
      orb = ORB.init( args, props );
```



```
    }

    protected boolean init(BancoDados client) {
       if (client == null) return false;
       try {
          // cria objeto
          bdc = new QuadroAvisosImpl(client); // 1a
          // exporta a referencia do objeto
          orb.connect(bdc);  // 1b
          // registra objeto no sistema de nomes
          org.omg.CORBA.Object objRef =
               orb.resolve_initial_references("NameService");  // 2
          NamingContext nctx = NamingContextHelper.narrow(objRef); // 3

          NameComponent ncomp = new NameComponent(NOMESERV, ""); // 4
          NameComponent[] raiz = {ncomp};
          nctx.rebind(raiz, bdc); // 5

          // imprime referencia do objeto no formato String (opcional)
          System.out.println(orb.object_to_string(bdc));
          System.out.println("Servidor no ar. Nome do servico: \'"+
                             NOMESERV +"\'");

          // espera requisicoes de objetos
          Thread waiting = new Thread(new Runner());
          waiting.start();
          return true;
       } catch(Exception e) { // (...)
       }
    }
    (...)
    public static void main(String[] args) {
       new CorbaDadosServerUI(args);
    }
}
```

Registrar um objeto remoto com CORBA exige um certo trabalho, já que os objetos são registrados no servidor de nomes dentro de um contexto hierárquico. O primeiro passo é inicializar o ORB e exportar o objeto para ele (linhas 1a e 1b acima). Depois, é preciso registrar o objeto em um sistema de nomes para que possa ser localizado por clientes remotos. É preciso encontrar um serviço chamado "`NameService`" no ORB e recuperar a referência `NamingContext` que ele retorna (linhas 2 e 3). Esta referência é necessária para termos acesso aos métodos do serviço de nomes será usado.

O servidor de nomes permite vários níveis de hierarquia. Como só temos um objeto a registrar, o criaremos na raiz (linha 4) de um componente de nome. Na linha 5, usamos o contexto de nomes para ligar este componente ao objeto.

Depois de implementado o servidor acima, o banco de dados torna-se acessível via CORBA,. Veja no capítulo 4 como executar a aplicação servidora.

Implementação do Cliente

O cliente CORBA precisa localizar o objeto no servidor através do nome que este último usou para se registrar. Não precisa saber onde está o objeto, mas precisa saber pelo menos o endereço de um servidor de nomes que possa localizá-lo. Obtendo-se uma referência ao objeto remoto, este precisa ser convertido no seu formato original. Depois disso, usá-lo é uma tarefa trivial. O acesso a métodos remotos é idêntico a invocações de métodos locais.

O cliente é mais uma implementação de `bancodados.BancoDados` e portanto pode ser usado por qualquer interface usuário. O código pode ser encontrado no diretório `bancodados/tier2/remote`. A listagem abaixo é parcial e mostra que, embora haja algum trabalho na obtenção da referência para o objeto remoto, a posterior invocação dos métodos remotos é trivial e igual à chamada de métodos locais.

```
public class CorbaClient implements BancoDados {

   public final static String NOMESERV = "anuncios";
   private QuadroAvisos bd;  // interface remota

   public CorbaClient(String[] args, Properties props) throws IOException {
      try {
         // inicializacao do ORB
         ORB orb = ORB.init(args, props);
         // registro no sistema de nomes
         org.omg.CORBA.Object objRef =
                   orb.resolve_initial_references("NameService");
         NamingContext nctx = NamingContextHelper.narrow(objRef);
         NameComponent ncomp = new NameComponent(NOMESERV, "");
         NameComponent[] raiz = {ncomp};
         org.omg.CORBA.Object obj = nctx.resolve(raiz);

         // Estreitamento do tipo
         bd = (QuadroAvisos)QuadroAvisosHelper.narrow(obj);
      } catch (Exception e) { // (...)
      }
```



```
(...)
   public void addRegistro(String texto, String autor)  {
      try {
         bd.addRegistro(texto, autor);
      } catch (ErroES e) {
         e.printStackTrace();
      }
   }
   (...)
}
```

Implantação do serviço

Para implantar o serviço, usando Java 2, é preciso executar primeiro o servidor de nomes:

```
tnameserv & (ou start tnameserv no Windows)
```

Depois, já pode-se rodar o servidor:

```
java bancodados.tier3.corba.CorbaDadosServerUI
```

Depois que a aplicação servidora estiver no ar, pode-se escolher o tipo de banco de dados que será disponibilizado aos clientes e selecionar a opção de menu "Conectar", informando o arquivo ou URL JDBC. Quando a aplicação informar que está a espera de clientes, estará pronta para receber requisições dos clientes CORBA.

Glossário

CORBA – *Common ORB Architecture*. Arquitetura Comum do ORB. Especificação que estabelece uma estrutura comum de comunicação entre aplicações independente da plataforma onde residem e da linguagem em que foram escritas.

OMA – *Object Management Architecture (OMA)*. Arquitetura de Gerência de Objetos. É uma representação de alto nível de um ambiente distribuído. Constitui-se de quatro componentes. ORBs, *Object Services* (objetos da aplicação), *Application Objects* (objetos da aplicação) e *Common Facilities* (recursos comuns).

OMG – *Object Management Group*. Grupo de Gerência de Objetos. Consórcio sem fins lucrativos criado em 1989 com o objetivo de promover a teoria e a prática da engenharia de software orientada a objetos, oferecendo uma arquitetura comum para o desenvolvimento de aplicações independente de plataformas de hardware, sistemas operacionais e linguagens de programação.

ORB – *Object Request Broker*. Corretor de Requisição de Objetos. Barramento comum que conduz as informações trocadas entre objetos.
IDL – *Interface Definition Language*. Linguagem de Definição de Interfaces. Define os tipos dos objetos, especificando suas interfaces[49]. A IDL da OMG tem uma sintaxe derivada do C++, sem as partes usadas na implementação de sistemas e acrescentando novas palavras-chave necessárias à especificação de sistemas distribuídos. Um mesmo IDL pode ser usado na geração de *stubs* e *skeletons* de linguagens diferentes, permitindo a comunicação de aplicações implementadas em linguagens distintas.
IIOP – *Internet Inter ORB Protocol*. Protocolo Internet Inter-ORB. Protocolo que estabelece pontes de comunicação entre ORBs diferentes em redes TCP/IP.
Stub. *Proxy* do cliente. Código intermediário que traduz e adapta as operações requisitadas pelo cliente ao formato de comunicação do ORB.
Skeleton. Esqueleto do servidor. Faz o trabalho inverso do *stub*. Transforma a requisição conuzida pelo ORB nos tipos nativos da implementação do servidor.
tnameserv – *Transient Name Server*. Servidor de nomes fornecido pelo JDK1.2. Deve ser executado na máquina servidora antes que qualquer servidor de objetos seja iniciado. Cada servidor de objetos deve registrar seu serviço no servidor de nomes.
idltojava – Compilador IDL fornecido pelo JDK1.2. Deve receber um arquivo IDL como entrada. Gera os *stubs*, *skeletons* para cada objeto declarado na IDL e cria pacotes e código auxiliar necessários ao desenvolvimento de clientes ou servidores de objetos CORBA em Java.

7.5. RMI

CORBA é uma solução para a comunicação entre aplicações escritas em Java ou em outras linguagens. Quando os dois lados falam Java, uma solução de desenvolvimento bem mais simples é RMI (*Remote Method Invocation*). RMI é 100% Java mas tem uma arquitetura muito parecida com CORBA. Este capítulo é uma versão do capítulo anterior, apresentando os mesmos exemplos, porém usando desta vez RMI em vez de CORBA.

Com RMI, um programa cliente poderá invocar um método em um objeto remoto da mesma maneira como faz com um método de um objeto local. Todos

os detalhes da conexão de rede são ocultados, permitindo que o modelo de objetos tenha sua interface pública mantida através da rede, sem precisar expor detalhes irrelevantes a ele, como conexões, portas ou endereços.

Há duas maneiras de usar RMI em Java. Uma é através de um protocolo chamado JRMP - *Java Remote Method Protocol*. Outra maneira é através de IIOP – *Internet Inter-ORB Protocol*. A primeira opção é adequada a aplicações 100% Java. Já a segunda, pode ser usada em ambientes heterogêneos. A forma de desenvolvimento, porém, é bastante parecida.

Um objeto remoto é obtido, usando JRMP, através de um servidor de nomes especial chamado *RMI Registry*. Ele deve estar rodando no servidor e permite que os objetos publicamente acessíveis através da rede sejam referenciados através de um nome. A classe `java.rmi.Naming` possui um método de classe `lookup()` que consulta um servidor de nomes RMI e obtém uma instância de um objeto remoto que poderá usar para invocar seus métodos.

Por exemplo, um determinado servidor de nomes RMI, situado na máquina `pleione.taurus.net` possui um registro chamado "Oceano" para um servidor de objetos remotos. Este servidor define os métodos `atira(x,y)` e `mapeia(x,y)`, numa interface `Territorio` ambos publicamente acessíveis. Para usar este objeto, pode-se fazer o seguinte:

```
Territorio mar;
mar = (Territorio)Naming.lookup("rmi://pleione.taurus.net/Oceano");
```

Com esses passos, obtemos um objeto `mar`, do tipo `Território`. O método `lookup` devolve `Object`, daí a necessidade da transformação em `Território`. Agora é possível invocar métodos remotos de `mar`:

```
tentativa[i] = mar.atira("C", 9);
```

É assim: tão simples quanto chamar um método local. O registro de nomes de objetos e sua posterior localização também é extremamente simples, já que não existe a organização em hierarquias de nomes como em CORBA.

Nos trechos de código acima, omitimos alguns detalhes, como a necessidade de tratar da exceção `RemoteException`. Este e outros detalhes sobre como usar o RMI serão ilustrados mais adiante.

Como funciona o RMI

Quando este trabalho foi iniciado, só havia uma forma de usar RMI: sobre os protocolos nativos Java. Pouco antes deste trabalho ser concluído, foi liberado aos desenvolvedores a primeira versão preliminar de RMI sobre IIOP (o mesmo protocolo usando nas comunicações CORBA). Incluímos neste capítulo algumas seções que irão destacar diferenças entre o RMI tradicional, chamado RMI sobre *Java Remote Method Protocol* (JRMP) e o novo RMI sobre IIOP.

RMI sobre JRMP

RMI/JRMP tem um funcionamento semelhante a CORBA, porém é muito mais simples e limitado à linguagem Java. É ideal para invocar objetos remotos quando tanto o cliente quanto o servidor são programas escritos em Java.

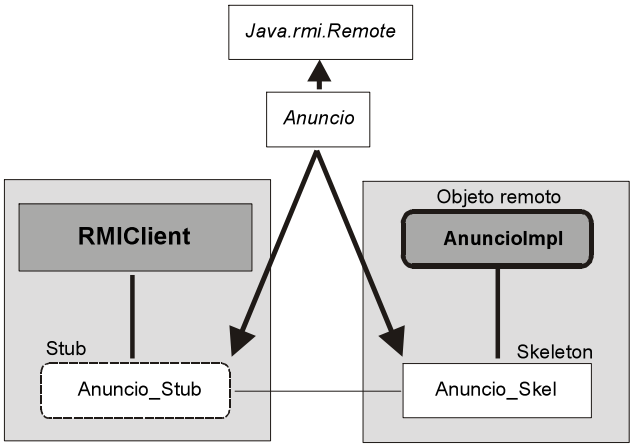


Figura 7-5

Para usar RMI, é necessário criar os objetos do servidor, definir um arquivo de interface Java implementado pela classe que define os objetos do servidor e rodar um compilador especial (análogo a um compilador IDL) que irá gerar os *stubs* e *skeletons* (figura 7-5). Um servidor de registro de nomes faz um papel semelhante ao repositório de interfaces CORBA, mantendo um registro dos objetos remotos disponíveis naquela máquina. Os clientes usam os arquivos de *stub* , em máquinas remotas, e passam a ter acesso aos objetos no servidor.

Além as classes da API, localizadas no pacote `java.rmi`, a plataforma Java 2 vêm com alguns aplicativos específicos para o RMI. O aplicativo `rmic` é o compilador gerador de *stubs* e *skeletons*. Ele deve ser rodado nas classes que implementam uma interface contendo os métodos acessíveis remotamente. Outro aplicativo é o `rmiregistry`. Ele deve estar rodando em cada máquina que tiver objetos remotos e desejar disponibilizá-los.

RMI sobre IIOP

Em dezembro de 1998 a *JavaSoft* liberou uma versão beta de RMI sobre IIOP. Esta nova arquitetura une a facilidade de desenvolvimento RMI com a portabilidade de CORBA. Objetos remotos RMI construídos de acordo com essa arquitetura são também objetos remotos CORBA. A nova ferramenta `rmic` distribuída com o pacote pode ser usada para gerar interfaces IDL a partir de implementações Java. As interfaces IDL geradas podem ser compiladas em outras linguagens.

A seção seguinte será, quase toda dedicada a RMI sobre JRMP. Porém, no final será mostrada uma maneira de realizar a conversão de aplicações RMI/JRMP para aplicações RMI/IIOP.

7.6. Construção de uma aplicação distribuída com RMI

Como nas seções anteriores, criaremos uma camada intermediária para o acesso remoto da aplicação de banco de dados. Nesta seção, usaremos RMI (sobre JRMP).

Interface Remote

O primeiro passo é escrever a interface pública que será acessível através da rede. Assim como na versão CORBA, teremos duas interfaces: `QuadroAvisos` e `Anuncio`. A interface RMI é uma interface Java e deve estender a interface `java.rmi.Remote`. Esta interface não têm novos métodos a serem implementados. É apenas uma indicação de que a nossa classe é uma interface RMI. Compare as interfaces com a versão IDL mostrada na última seção.

Classe bancodados/tier3/rmi/Anuncio.java

```
package bancodados.tier3.rmi;

import java.util.Date;
import java.rmi.Remote;
import java.rmi.RemoteException;

   public interface Anuncio extends Remote {
   public int getNumero() throws RemoteException;
   public String getTexto()  throws RemoteException;
   public Date getData()  throws RemoteException;
   public String getAutor() throws RemoteException;
}
```


Classe bancodados/tier3/rmi/QuadroAvisos.java

```
package bancodados.tier.rmi;

import java.rmi.Remote;
import java.rmi.RemoteException;
import bancodados.server.RegistroInexistente;

public interface QuadroAvisos extends Remote {

   public int length() throws RemoteException ;

   public Anuncio getRegistro(int numero)
       throws RegistroInexistente, RemoteException ;

   public void addRegistro(String texto, String autor)
       throws RemoteException;
(...)
}
```

Diferentemente de CORBA, em RMI não compilamos as interfaces. Precisamos *implementá-las* primeiro em classes que definem objetos remotos para só depois usar o gerador de *stubs* e *skeletons*.

Para que funcione como objeto remoto, uma classe deve estender a classe da API `java.rmi.server.UnicastRemoteObject`. Deve também implementar a sua interface `Remote` correspondente. Os métodos do objeto remoto todos podem provocar `java.rmi.RemoteException` e sua ocorrência deve ser declarada (ou tratada) em cada um dos métodos definidos nas subclasses de `UnicastRemoteObject`. Não somente os métodos. O construtor do objeto remoto também pode provocar uma exceção quando o objeto estiver sendo criado, portanto, mesmo que o construtor seja vazio, ele precisa ser definido mesmo que só para acrescentar a declaração "`throws java.rmi.RemoteException`".

Implementamos as interfaces acima nos arquivos `AnuncioImpl.java` e `QuadroAvisosImpl.java`. A listagem abaixo mostra a implementação parcial de `QuadroAvisosImpl`:

```
package bancodados.tier3.rmi;

import java.rmi.*;
import java.rmi.server.*;
import java.util.Date;

import bancodados.*;
```



```
public class QuadroAvisosImpl extends UnicastRemoteObject
                              implements QuadroAvisos {

   private BancoDados fonte;       // acesso a operacoes de cliente local

   public QuadroAvisosImpl(BancoDados fonte) throws RemoteException {
      this.fonte = fonte;
   }

   public int length() throws RemoteException {
      return fonte.length();
   }


   public void addRegistro(String texto, String autor) throws RemoteException {
      fonte.addRegistro(texto, autor);
   }

   public boolean setRegistro(int numero, String texto, String autor)
      throws RegistroInexistente, RemoteException  {
      return fonte.setRegistro(numero, texto, autor);
   }
(...)
```

Observe que para a implementação, não interessa a forma de armazenamento do banco de dados porque novamente, esta aplicação usa a interface `bancodados.BancoDados`..

O próximo passo é a geração de *stubs* e *skeletons*, mas antes, vamos tratar da criação do servidor que irá servir os objetos aos clientes remotos.

Implementação do servidor

Para colocar os objetos remotos "no ar" e permitir que aplicações cliente remotas tenham acesso a eles, é preciso construir um servidor. O servidor RMI é uma aplicação *middleware* que vai criar o objeto remoto (`QuadroAvisosImpl`, por exemplo), que por sua vez vai acessar, como cliente, a terceira camada da aplicação.

Implementamos o servidor como uma subclasse de `DadosServerFrame`. O método `init()` do servidor tem como principal objetivo criar uma instância do objeto remoto e registrá-lo no serviço de nomes: o *RMI Registry*. Para registrar o nome no *RMI Registry*, usamos o método de classe `Naming.rebind`. Veja o código (parcial) do servidor RMI `RMIDadosServerUI`:


```
package bancodados.user;

import java.io.*;
import java.rmi.*;

import bancodados.*;
import bancodados.tier3.rmi.*;

(...)
public class RMIDadosServerUI extends DadosServerFrame {

   /** Constante com nome do serviço */
   public final static String NOMESERV = "anuncios";

   private static String frameTitle = "Servidor RMI";  // Titulo da aplicação
   private BancoDados client;
   private QuadroAvisosImpl bdrmi;      // implementação do servidor

   public RMIDadosServerUI() {
      super(frameTitle);
   }

   protected boolean init(BancoDados client) {
      if (client == null) return false;
      try {

          // cria objeto
         bdrmi = new QuadroAvisosImpl(client);
         // exporta a referencia e registra objeto no sistema de nomes
         Naming.rebind(NOMESERV, bdrmi);
         // servidor no ar aguardando conexoes
         System.out.println("Servidor no ar. Nome do servico: \'"
                            + NOMESERV +"\'");
         return true;
      } catch(Exception e) {        // (...)
      }
   }
(...)
public static void main(String[] args) {
      new RMIDadosServerUI();
   }
}
```


Geração dos Stubs e Skeletons

Antes de gerar os *stubs* e *skeletons*, é necessário compilar as classes do servidor.

```
javac AnuncioImpl.java
javac QuadroAvisosImpl.java
```

Em seguida, executa-se o `rmic` (gerador de *stubs* e *skeletons*) sobre a classe compilada do servidor:

```
rmic –d /jad/apps/ bancodados.tier3.rmi.AnuncioImpl
rmic –d /jad/apps/ bancodados.tier3.rmi.QuadroAvisosImpl
```

Esta operação irá criar quatro novos arquivos. Dois arquivos para cada implementação. Um é o *stub* do cliente. O outro, o esqueleto do servidor. Eles devem estar no mesmo pacote (diretório) que as classes `AnuncioImpl` e `QuadroAvisosImpl`:

```
QuadroAvisosImpl_stub.class
QuadroAvisosImpl_skel.class

AnuncioImpl_stub.class
AnuncioImpl_skel.class
```

Desenvolvendo o cliente

O cliente é muito simples. Sua principal função é localizar o objeto. Depois disso, é só usar a referência. Uma instância do objeto remoto é obtida usando o método `lookup()`, que consulta o *RMI Registry* do servidor escolhido e retorna um objeto de acordo com a URL definida. O cliente é mais uma implementação da interface `bancodados.BancoDados` (pode ser usado por qualquer interface do usuário). A listagem abaixo é parcial.

```
package bancodados.tier2.remote;

import java.io.IOException;
import java.util.Date;
import java.rmi.*;

import bancodados.*;
import bancodados.tier3.rmi.*;

public class RMIClient implements BancoDados {

   public final static String NOMESERV = "anuncios";

   private QuadroAvisos bd;
```



```
   public RMIClient(String hostname) throws IOException {

      try {
        Object obj = Naming.lookup("rmi://" + hostname + "/" + NOMESERV);
        bd = (QuadroAvisos) obj;

      } catch (Exception e) { // (...)
      }

   }

   public void addRegistro(String texto, String autor)  {
      try {
         bd.addRegistro(texto, autor);
      } catch (RemoteException e) { // (...)
      }
   }

(...)
```

Inicialização do serviço de nomes

Para implantar o serviço, deve-se iniciar o *RMI Registry* a máquina que será servidora. Este é o verdadeiro servidor de objetos que fica escutando a porta 1099 (*default*) onde clientes podem conectar e tentar descobrir quais objetos remotos estão disponíveis.

```
rmiregistry &       (Unix)
start rmiregistry  (WinNT/95/98)
```

Estando o *RMI Registry* no ar, pode-se executar o servidor

```
java bancodados.tier3.rmi.RMIDadosServerUI
```

Para iniciar o servidor, escolha uma fonte de dados (arquivo ou JDBC) e depois selecione a opção de menu "Conectar". Quando o servidor informar que o objeto foi exportado, estará pronto para receber requisições de clientes.

Conversão de uma aplicação RMI/JRMP em RMI/IIOP

São poucas as mudanças necessárias para transformar uma aplicação JRMP (que só usa objetos remotos escritos em Java) em aplicação IIOP (que pode usar qualquer objeto remoto CORBA) são realizadas principalmente nas classes que implementam clientes e servidores. Os objetos remotos precisam mudar a classe de quem herdam. Não é preciso mexer nas interfaces `Remote`.

Os novos objetos remotos devem estender **javax.**rmi.**PortableRemoteObject** e não mais **java.**rmi.server.**UnicastRemoteObject**:

```
public class QuadroAvisosImpl extends PortableRemoteObject
                               implements QuadroAvisos { ... }

public class AnuncioImpl extends PortableRemoteObject
                          implements Anuncio { ... }
```

Os servidores não devem usar o serviço de nomes do *RMIRegistry* (que usa JRMP). Deve usar o serviço de nomes nativo do Java (pacote `java.naming`) para registrar objetos remotos RMI em um ORB:

```
import javax.rmi.*;     // RMI sobre IIOP
import javax.naming.*;  // JNDI (Naming services)

public class RMIIIOPDadosServerUI extends DadosServerFrame {

(...)
   private QuadroAvisosImpl bdrmi;      // implementação do servidor
   private Context namingContext;       // ambiente RMI-IIOP

   public RMIIIOPDadosServerUI() {
      super(frameTitle);
      try {
         Hashtable env = new Hashtable();
         env.put("java.naming.factory.initial",
                 "com.sun.jndi.cosnaming.CNCtxFactory");
         env.put("java.naming.provider.url",
                 "iiop://localhost:900");   // nameserver ñ precisa ser local!

         // obtem contexto de serviço de nomes
         namingContext = new InitialContext(env);
      } catch (NamingException e) {
         e.printStackTrace();
      }
   }

   protected boolean init(BancoDados client) {
      try {
         bdrmi = new QuadroAvisosImpl(client);
         // exporta a referencia e registra objeto no sistema de nomes
         namingContext.rebind(NOMESERV, bdrmi);
         System.out.println("Servidor no ar. Nome do servico: \'"
                            + NOMESERV +"\'");
         return true;
      } catch(Exception e) { // (...)
      }
```



```
    }
```

O cliente deve fazer o mesmo. Obter um contexto, localizar o objeto, obter a referência.

```
import javax.rmi.*;     // PortableRemoteObject
import javax.naming.*;  // pacote JNDI (nomes)

public class RMIIIOPClient implements BancoDados {
   public final static String NOMESERV = "anuncios";
   private QuadroAvisos bd;

   public RMIIIOPClient(Hashtable env) throws IOException {
       try {
         // Obtenção do contexto inicial (JNDI) usando por RMI sobre IIOP
         Context namingContext = new InitialContext(env);
         // obtem objeto via sistema de nomes (usando JNDI)
         Object obj = namingContext.lookup(NOMESERV);
         // Estreita (converte) o tipo
         bd = (QuadroAvisos)PortableRemoteObject.narrow(obj,
                                                QuadroAvisos.class);

      } catch (Exception e) { // (...)
      }
   }

   public void addRegistro(String texto, String autor)  {
      try {
         bd.addRegistro(texto, autor);
      } catch (RemoteException e) {
         e.printStackTrace();
      }
   }
```

Para gerar os stubs e skeletons para o RMI/IIOP, é preciso rodar o novo `rmic` com a opção `-iiop`:

```
rmic -iiop -d /jad/apps/ bancodados.tier3.riiop.AnuncioImpl
rmic -iiop -d /jad/apps/ bancodados.tier3.riiop.QuadroAvisosImpl
```

Isto deve gerar quatro arquivos dentro do pacote `bancodados.tier3.riiop`:

`QuadroAvisos_Stub.class`	- *Stub*
`QuadroAvisosImpl_Tie.class`	- *Skeleton*
`Anuncio_Stub.class`	- *Stub*
`AnuncioImpl_Tie.class`	- *Skeleton*

Para iniciar o servidor, é preciso que o serviço de nomes do JNDI esteja no ar:

`tnameserv & (ou start tnameserv).`

Depois, pode-se executar o servidor como foi feito com as outras tecnologias.

Resumo

Os passos a seguir resumem o processo de desenvolvimento e implantação de um servidor de objetos usando RMI/JRMP:

- 1. Definir a interface: declare todos os métodos que serão acessíveis remotamente em um arquivo de interface Java que estende `java.rmi.Remote`. Todos os métodos devem declarar `throws java.rmi.RemoteException`. Isto deve ser feito para cada objeto que será acessível através da rede.
- 2. Escrever a implementação dos objetos remotos - classes que estendem `java.rmi.server.UnicastRemoteObject` e que implementam a interface criada no passo 1. Todos os métodos devem declarar `throws java.rmi.RemoteException` inclusive o construtor, mesmo que seja vazio.
- 3. Estabelecer um gerente de segurança (`SecurityManager`[1]), obter uma instância do objeto a ser servido e implementar o mecanismo de registro do objeto do *RMIRegistry* no servidor.
- 4. Compilar o código-fonte dos objetos remotos com `javac`, gerando um arquivo de classe do servidor.
- 5. Compilar o arquivo de classe dos objetos remotos com `rmic`, gerando um arquivo stub, e um arquivo skeleton. (`ObjImpl_stub.class` e `ObjImpl_skel.class`)
- 6. Escrever, compilar e instalar o(s) cliente(s)
- 7. Instalar o stub no(s) cliente(s) (copiar o stub gerado no passo 5 para as máquinas e CLASSPATH acessíveis pelos clientes ou prover mecanismo que permita ao cliente fazer o download do stub.)
- 8. Iniciar o *RMI Registry* no servidor
- 9. Iniciar o servidor de objetos
- 10. Iniciar os clientes informando o endereço do servidor.

---

[1] No nosso exemplo não definimos uma instância de `RMISecurityManager`. Sem um `SecurityManager` não é permitido a um cliente carregar stubs e outras classes dinamicamente do servidor. Para instalar um `RMISecurityManager`, faça: `System.setSecurityManager(new RMISecurityManager());`

Glossário

RMI - *Remote Method Invocation*. Invocação Remota de Método.
Stub. *Proxy* do cliente. Código intermediário que traduz e adapta as operações requisitadas pelo cliente em formato de dados adequado à transmissão via rede.
Skeleton. Esqueleto do servidor. Faz o trabalho inverso do *stub*. Repassa a requisição para o servidor.
rmic – *RMI Compiler*. Compilador (gerador) de *stubs* e *skeletons* fornecido pelo JDK1.2. Deve receber o objeto remoto (que implementa uma subinterface de `java.rmi.Remote`) como entrada (código compilado). Gera os *stubs*, *skeletons* para cada objeto.
rmiregistry – Servidor de nomes RMI fornecido pelo JDK1.2. Deve ser executado na máquina servidora antes que qualquer servidor de objetos seja iniciado. Cada servidor de objetos deve registrar seu serviço no servidor de nomes. Depende da existência de um servidor ou serviço de nomes (DNS ou arquivo de *hosts*) TCP/IP.

7.7. Comparação entre as tecnologias

Esta seção discute a utilização de aplicações baseadas nas tecnologias estudadas, levando em conta resultados obtidos em testes de desempenho e cenários onde cada tecnologia pode ser aplicada.

Objetos remotos (RMI/CORBA) *versus* Sockets TCP/IP

Toda a comunicação em rede TCP/IP entre uma aplicação Java e outra aplicação ou serviço pode ser realizada através do pacote `java.net`. Soquetes (classe `java.net.Socket`) implementam uma conexão confiável usando o protocolo TCP. Datagramas UDP (classe `java .net.DatagramSocket`) implementam conexões mais eficientes, porém não confiáveis. Utilizando essas e outras classes do pacote `java.net`, pode-se implementar qualquer tipo de interoperabilidade entre aplicações TCP/IP. Por que então utilizar tecnologias de objetos remotos, que observamos ser menos eficientes?

Se o objetivo da aplicação é simplesmente transferir comandos ou informações de um lugar para outro, usar o pacote `java.net` pode ser a melhor opção. Usar Java em rede é quase tão simples quanto usar Java para operações locais de

entrada e saída. Talvez seja necessário usar *threads*. Uma típica aplicação cliente-servidor terá provavelmente alguns serviços em *threads* diferentes (por exemplo, para que um servidor possa aceitar múltiplos clientes).

Mas se a aplicação precisa criar objetos na aplicação remota ou invocar métodos remotos que retornam objetos, a programação usando `java.net` poderá se tornar bastante complexa. "Os requisitos essenciais em um sistema de objetos distribuídos são a capacidade de criar ou invocar objetos em um processo ou máquina remota, e interagir com eles como se fossem objetos do processo e máquina correntes"[FARL98]. Para criar um objeto remoto, o cliente precisa saber qual a sintaxe usada no servidor para realizar tal tarefa. O servidor também precisará saber como retornar uma informação para o cliente. Portanto, será necessária a existência de algum protocolo de mensagens para "enviar requisições para agentes remotos para que criem novos objetos, para invocar métodos nesses objetos e para eliminar os objetos quando não mais precisarmos deles"[FARL98].

Se a linguagem é Java e um cliente desejar criar um objeto remoto, ele precisará instruir o servidor a carregar a classe correspondente (pelo nome), precisará passar os argumentos do construtor do objeto correspondente, realizar uma operação de criação de objeto (usando `new`) e retornar para o cliente uma referência remota ao objeto criado. Com esta referência, o cliente poderá invocar métodos remotos e quando não precisar mais do objeto, informar ao servidor que o mesmo pode ser destruído.

O servidor também precisará manter um controle sobre os objetos criados remotamente. Se um outro cliente tenta criar um objeto já criado, o servidor deverá retornar-lhe uma referência do objeto existente. Se um dos clientes solicitar a remoção de um objeto, o servidor precisará descobrir se ainda há outras referências remotas para aquele objeto. O servidor precisa manter mais que uma tabela de objetos criados. Precisa também mapear cada um dos métodos daqueles objetos e seus argumentos, que podem, por sua vez, ser também objetos remotos.

É possível desenvolver um mecanismo desses para uma determinada aplicação distribuída usando somente TCP/IP (`java.net`). Ela provavelmente será mais eficiente que uma solução usando tecnologias de objetos distribuídos. O preço pela eficiência será cobrado quando for necessário alterar alguma característica da aplicação. Por utilizar um protocolo proprietário, quase não há reutilização de

componentes e é provável que vários procedimentos tenham que ser rescritos. Para, por exemplo, acrescentar um método em um objeto remoto, será necessário mexer em todas as estruturas que usam o objeto remoto em questão, e também naquelas que possuem métodos ou construtores que recebem ou retornam o objeto remoto.

Poderíamos melhorar o protocolo proprietário, acrescentando operações genéricas, independentes dos detalhes da implementação. Isto certamente diminuiria a sua eficiência, mas permitiria um desenvolvimento mais simples, possibilitando talvez uma maior reutilização. Continuando a desenvolver o protocolo dessa maneira, terminaríamos por reinventar um mecanismo como o RMI: mais fácil de desenvolver e manter, porém menos eficiente.

As tecnologias de objetos remotos, como RMI, e CORBA oferecem uma infraestrutura que permite invocações de operações em objetos localizados em máquinas remotas como se fossem locais à aplicação que os usa [VOGE98]. Tanto RMI e CORBA são implementadas em Java usando esses recursos de baixo nível (`java.net`), mas escondem a complexidade tornando o desenvolvimento em rede quase tão simples quanto o desenvolvimento local. O preço é a menor eficiência, uma vez que a tecnologia tem que lidar com uma gama de possibilidades nem sempre presentes na aplicação que as utiliza. Tanto RMI e CORBA geralmente requerem uma infraestrutura adicional, externa à aplicação, como sistemas de nomes e gerentes de objetos, para manter o controle sobre os objetos remotos. Essas estruturas, além de ocuparem mais recursos e memória do sistema, impõem obstáculos adicionais que os dados precisam transpor, antes de serem enviados pela rede através de conexões TCP ou UDP.

Portanto, embora o uso direto de TCP/IP em aplicações Java permita obter o melhor desempenho, a complexidade da aplicação e a necessidade de manutenção futura podem induzir à opção por uma tecnologia de objetos distribuídos, como RMI.

RMI/JRMP versus RMI/IIOP e CORBA

A tecnologia RMI, nativa do Java 2, utiliza para a comunicação um protocolo chamado *Java Remote Method Protocol* (JRMP), que permite a criação de objetos remotos e a chamada de métodos remotos, passando parâmetros e retornando valores e referências de forma transparente ao programador. Através do RMI, o

programador invoca métodos remotos da forma como invoca métodos locais. A comunicação entre os clientes e os objetos remotos é centralizada no *RMI Registry* – programa rodando no servidor que atua tanto como servidor de nomes como gerente de objetos Java [SUN96]. O *RMI Registry* roda sobre a máquina virtual Java local e permite que ela se comunique com uma máquina virtual remota.

O suporte à tecnologia CORBA foi introduzido na plataforma Java 2 através de um conjunto de pacotes de prefixo `org.omg`. Embora parecidos na superfície, RMI e CORBA são bastante diferentes quando sua estrutura interna é analisada. RMI é uma tecnologia nativa a Java e é, em essência, uma extensão ao núcleo da linguagem. CORBA é uma tecnologia de *integração*, independente de Java. Seu objetivo é unir tecnologias de *programação* incompatíveis entre si [CURT97]. A comunicação em CORBA não ocorre diretamente entre o cliente e o servidor, ou entre máquinas virtuais, mas entre ORBs (*Object Request Brokers*) – gerentes de objetos presentes nas máquinas que possuem clientes ou servidores. Em redes TCP/IP, CORBA geralmente utiliza o protocolo IIOP – *Internet Inter-ORB Protocol*, que é responsável pela comunicação entre ORBs. As comunicações não são centralizadas (como são no *RMI Registry*) e os objetos podem estar distribuídos pela rede [FARL97][OMG98].

O "denominador comum" nas comunicações JRMP é uma interface escrita em Java. Nas comunicações IIOP, o denominador comum é uma interface IDL (neutra com respeito à linguagem), que permite a possibilidade de comunicação com objetos escritos em outras linguagens.

Ainda em versão beta (como extensão do Java 2), a nova versão de RMI suporta a comunicação através de IIOP – *Internet Inter-ORB Protocol*, que é o protocolo utilizado na comunicação entre objetos remotos CORBA. Na prática o novo RMI *é* CORBA pois só se parece com RMI na superfície. Usa a tecnologia de *programação* Java, mas está de acordo com a tecnologia de *integração* CORBA. O programador pode desenvolver em Java, como já fazia com o RMI antigo. Pode usar suas classes para criar objetos remotos que serão utilizados via JRMP ou IIOP. O pacote contém ainda ferramentas que geram IDL a partir de Java, facilitando a comunicação entre objetos de linguagens diferentes. Esta nova solução permite que programas Java escritos em RMI possam se comunicar com objetos remotos CORBA e vice-versa.

Agora com três opções de objetos distribuídos, qual delas devemos usar? RMI sobre JRMP, CORBA ou RMI sobre IIOP? Novamente, não temos a intenção de apontar esta ou aquela tecnologia como a melhor. RMI (JRMP) é uma alternativa viável para um conjunto de aplicações. CORBA (incluindo RMI sobre IIOP), é uma tecnologia apropriada para outro conjunto de aplicações. Existem aplicações que podem ser desenvolvidas usando qualquer uma das duas tecnologias, mas há muito mais aplicações onde uma das duas possui vantagens distintas sobre a outra [CURT97].

Usar RMI/JRMP em vez de CORBA/IIOP é o ideal quando todas as partes da aplicação distribuída são escritas em Java. O desempenho da aplicação provavelmente será melhor (com base nos resultados dos experimentos que realizamos) e é possível utilizar recursos da linguagem Java que não são disponíveis via CORBA, como carregamento de classes remotas (para serem instanciadas localmente) e objetos serializados [FARL98]. Além do desempenho, uma vantagem do RMI/JRMP sobre CORBA/IIOP é o desenvolvimento mais simples. Isto, antes do advento do RMI sobre IIOP:

- Com RMI os tipos dos objetos são preservados. É preciso realizar conversões adicionais em CORBA. Usando RMI/IIOP as conversões são feitas de forma transparente.
- RMI é Java. Não é necessário aprender uma nova linguagem. É preciso usar IDL para definir as interfaces dos objetos remotos em CORBA. Usando RMI/IIOP o IDL é gerado automaticamente, não sendo, portanto, necessário conhecer a linguagem.
- Uma única linha de código é necessária para registrar ou localizar um objeto no *RMI Registry*. CORBA possui um serviço de registro de nomes mais sofisticado que exige a obtenção de várias referências para registrar até mesmo um único objeto. RMI sobre IIOP exige um esforço menor que CORBA, mas ainda bem maior que RMI sobre JRMP (o registro de objetos é a "parte CORBA" da tecnologia).

Usar CORBA é a melhor opção quando partes da aplicação estão escritas em outra linguagem. RMI não suporta a comunicação com objetos que não sejam objetos Java. Para aplicações totalmente escritas em Java, observamos uma taxa de transferência menor nas aplicações CORBA. Muitas transformações extras são necessárias para atingir o "denominador comum" IDL, que uma interface Java.

Mas uma solução CORBA pode melhorar o desempenho de uma aplicação Java utilizando objetos escritos em linguagens mais eficientes (como C) para tarefas críticas. Objetos especificados em IDL podem ser substituídos por quaisquer outros escritos em outras linguagens sem que o restante da aplicação seja afetada [VOGE 98].

Há ainda, vantagens que podem levar à escolha de CORBA/IIOP sobre RMI/JRMP mesmo em ambientes somente Java:

- Os clientes RMI precisam saber *em que máquina* estão os objetos. O *RMI Registry* deve estar executando na máquina onde estarão os objetos remotos. Os objetos CORBA, por sua vez, podem estar em *qualquer lugar* da rede e o cliente não precisa saber da sua localização. A comunicação em redes TCP/IP é realizada entre ORBs de qualquer plataforma ou linguagem usando o protocolo IIOP.
- Enquanto RMI/JRMP suporta apenas nomes de objetos definidos na mesma raiz, o servidor de nomes Java usado em aplicações CORBA suporta a organização hierárquica de nomes de objetos, dentro de contextos que se assemelham a diretórios em um sistema de arquivos, evitando o risco de conflito de nomes. Isto é útil quando várias aplicações usam o mesmo registro de nomes ou registram muitos objetos.

Se uma aplicação Java distribuída não pretende interagir com código em outras linguagens e se a localização dos objetos em um local específico não for um problema, RMI pode ser a solução ideal de objetos remotos para essa aplicação. Mas é preciso ter cuidado, pois RMI é parte de Java e não uma tecnologia de integração independente. No futuro, certamente surgirão novas linguagens e os sistemas Java serão os próximos sistemas legados que exigirão integração com os novos sistemas, e RMI não será capaz de oferecê-la. CORBA, por ser uma tecnologia de integração independente de linguagem de programação é uma melhor defesa contra a obsolescência [CURT97].

Mas com a opção de usar tanto JRMP como IIOP (ainda em beta), RMI pode vir a ser a melhor opção para a implementação de objetos distribuídos usando Java, oferecendo ao mesmo tempo facilidade de desenvolvimento e portabilidade com independência de plataforma e linguagem, além de proteção contra a obsolescência futura. A troca de JRMP por IIOP é direta. A implementação dos objetos remotos só precisam mudar uma linha de código e todo o restante

do código é aproveitado, sem alterações. A única alteração maior ocorre no servidor que irá registrar os objetos remotos, já que não mais usará o *RMI Registry*, mas um ORB.

7.8. Resumo

Para organizar todas as informações apresentadas, construímos uma tabela comparativa, comparando diversos aspectos das tecnologias analisadas e mostrando também onde (em que pacote nativo ou extensão) cada tecnologia é fornecida. A tabela 7-5 compara tecnologias de objetos distribuídos. Essas tecnologias podem ser usadas na construção de camadas (*tiers*) adicionais entre o cliente e o servidor.

Tabela 7-5 – Quadro comparativo: tecnologias de objetos distribuídos e TCP/IP default.

Tecnologia de rede	Cliente (qualquer um) usando camada (*tier*) intermediária com tecnologia ...			
	Sockets TCP/IP	**RMI sobre JRMP**	**RMI sobre IIOP**	**CORBA sobre IIOP**
DESEMPENHO				
Taxa de transferência de dados[2]	Melhor possível.	Mais lento que TCP/IP.	Mais lento que RMI/JRMP.	Mesma ordem que RMI/IIOP.
DESENVOLVIMENTO, MANUTENÇÃO E REUSO				
Flexibilidade de desenvolvimento. Grau de flexibilidade: 1 a 5 (1 – nenhuma, 5 – total)	4	2	3	3
Facilidade de desenvolvimento[3] Grau de facilidade: 1 a 5 (1 – fácil a 5 – difícil)	5	3	3	4
Quantidade de código adicional a escrever (1 - pouco , 5 - bastante)	5	2	3	3
SUPORTE E DISPONIBILIDADE (PACOTES)				
Núcleo JDK 1.2 (Java 2)	java.net (núcleo)	java.rmi (núcleo)	javax.rmi (extensão)	org.omg (núcleo)
Núcleo JDK 1.1	java.net (núcleo)	java.rmi (núcleo)	-	org.omg (extensão)
Núcleo JDK 1.0	java.net (núcleo)	java.rmi (extensão)	-	-

Utilizar uma solução baseada no pacote `java.net` traz algumas vantagens: maior compatibilidade (por ser nativa ao núcleo da plataforma Java), flexibilidade

[2] As limitações da rede utilizada no estudo de caso correspondente não permitiram que se apresentassem conclusões mais genéricas (e quantitativas) quanto ao desempenho.

[3] Leva em consideração número de linhas de código adicionais (em relação a uma aplicação *standalone*) que precisam ser escritas, linguagens adicionais que precisam ser aprendidas (como IDL), necessidade de programar usando *threads*, necessidade de programar em baixo nível (conexões, *streams*, portas, etc.). Não leva em conta a necessidade de se aprender a API específica para cada tecnologia.

(permite conectividade com agentes escritos em outras linguagens) e velocidade. A complexidade da programação em baixo nível e a baixa reutilização de código são motivos que nos levam a procurar soluções de objetos distribuídos. Em estudos realizados com as três tecnologias de objetos distribuídos, comparamos três tecnologias diferentes: RMI sobre JRMP (nativa desde o JDK1.1), CORBA sobre IIOP (nativa desde o JDK1.2[4]) e RMI sobre IIOP (extensão ao JDK1.2 ainda em beta). Nos casos e ambientes estudados, a tecnologia RMI sobre JRMP mostrou-se a mais eficiente das três. Também é a mais fácil de usar, pois não exige conhecimentos de rede (portas, soquetes, etc.) ou de outras linguagens (IDL), e é a que impõe menos alterações no código de uma aplicação *standalone* para torná-la uma aplicação distribuída. Por outro lado, é preciso que todas as partes da aplicação tenham sido escritas em Java, e que os objetos remotos permaneçam em locais definidos.

As vantagens em se usar CORBA ou RMI sobre IIOP está na possibilidade de integração da aplicação Java com outras aplicações ou objetos escritos em outras linguagens. Mas CORBA exige que o programador conheça IDL e saiba usar os métodos da API para realizar as transformações entre tipos de dados CORBA e Java. RMI esconde essa complexidade e, através do protocolo IIOP, permite que a aplicação se comporte de forma idêntica a uma aplicação CORBA. Nos casos estudados, as tecnologias CORBA e RMI sobre IIOP tiveram desempenho (quanto à taxa de transferência de dados) equivalente. Tal resultado é uma contribuição importante, apesar das limitações do experimento, pois reflete um aspecto de uma tecnologia lançada há pouco tempo. No caso analisado, o programador poderia desenvolver usando RMI sobre IIOP em vez de CORBA, obtendo o mesmo desempenho e a integração da arquitetura CORBA a um custo menor de desenvolvimento.

[4] Plataforma Java 2

JAVA 8

Aplicações, Applets

NESTE MÓDULO É APRESENTADA A ARQUITETURA DOS APPLETS, os métodos de pintura, classes para imagens, sons e gráficos e como usar e criar applets.

Tópicos abordados neste módulo

- Applets
- Métodos e objetos para desenho gráfico

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- Construir um applet ou converter uma aplicação existente em um applet

- Saber identificar as limitações dos applets
- Usar de forma correta os métodos de pintura e o AWT thread
- Usar um clipe de áudio e importar uma imagem para um applet
- Fazer desenhos em um applet ou aplicação da AWT

8.1. Applets e Gráficos

APPLETS SÃO APLICAÇÕES EM JAVA que são iniciadas a partir de uma página Web. Geralmente usam o contexto gráfico da página para exibir e fazer alguma coisa, mas podem também abrir janelas e escrever na saída padrão (o "Java Console" dos browsers), embora não seja usual.

Applets também têm mais restrições que aplicações por razões de segurança. A maioria dos applets transferidos através da rede não pode:

- Ler, escrever, renomear, apagar, executar ou criar arquivos e diretórios em sistema local.
- Obter informações de arquivos locais como listar conteúdo de diretórios, verificar a existência de um arquivo, saber o tamanho, nome ou data de modificação de um arquivo.
- Criar uma conexão de rede para qualquer máquina a não ser aquela de onde veio a applet.
- Escutar ou aceitar conexões de rede em qualquer porta do sistema local.
- Obter quaisquer informações sobre o usuário que está logado, como userid, etc.
- Definir propriedades do sistema.
- Abortar o interpretador Java (`System.exit()` é ilegal em Applets).
- Usar classes que não pertençam aos oito pacotes da biblioteca padrão Java

Apesar de todas essas restrições, ainda há muito que se pode fazer com applets. Na versão 1.1 de Java, a Sun introduziu um modelo mais flexível de segurança baseado em certificados.

8.2. Como incluir uma applet numa página Web

As páginas que formam a Web são escritas em uma linguagem de descrição de texto chamada HTML. A linguagem HTML consiste de descritores que marcam o texto especificando a forma como o texto será formatado em um browser e marcando a posição onde serão incluídos objetos, como figuras, animações e conteúdo executável.

Descritor <applet>

O descritor `<applet>` é usado pelo HTML 3.2 como forma de incluir conteúdo executável em uma página. A applet aparecerá na parte da página onde for incluído o descritor. Existem atributos para alinhamento e redimensionamento que são idênticos aos utilizados para incluir imagens, através do descritor `<img>`.

A sintaxe mínima do descritor `<applet>` é a seguinte:

```
<applet code="nome_do_arquivo_class"
        height="altura_em_pixels"
        width="largura_em_pixels">
...
[parâmetros e opções para browsers incompatíveis]
...
</applet>
```

Os três atributos `code`, `height` e `width` são os únicos obrigatórios. Se a applet não estiver acessível através de um caminho de subdiretórios em relação ao documento que o referencia, é necessário especificar ainda um argumento `codebase="URL_do_diretório_base"` que contém a URL do lugar onde a applet se encontra. Outro atributo opcional é `name="nome_da_applet"` que rotula a applet com um nome.

Os outros atributos do descritor `<applet>`: `alt`, `align`, `vspace` e `hspace` são os mesmos do descritor HTML `<img>`.

Os parâmetros da applet são passados através do descritor `<param>`, que deve ficar entre `<applet ...>` e `</applet>` (ambos os descritores `<applet>` e `</applet>` devem ser usados, mesmo que não haja parâmetros). `<param>` não tem descritor de fechamento. Recebe dois atributos obrigatórios:

```
<param name="parâmetro" value="valor">
```


`name="parâmetro"` contém o nome de um parâmetro definido pelo programador da applet. `value="valor"` contém um valor que pode ser mudado pelo autor da página Web, mesmo que ele não tenha acesso à fonte da applet.

Qualquer coisa colocada entre `<applet>` e `</applet>`, que não esteja dentro de `<param>` é ignorada por browsers que suportam Java, mas aparecem nos que não a suportam, portanto, é um bom lugar para colocar imagens alternativas, links ou qualquer outra informação que informe à pessoa que lê a página, que ali deveria haver uma applet.

Incluíndo uma applet na página

Para incluir uma applet em uma página, não é necessário saber nada de Java, só um mínimo de HTML. A applet pode ser incluída em qualquer lugar onde possa ser incluída uma imagem.

Na Web há várias localidades que distribuem applets para uso público. Para incluí-los é só chamar a classe principal no descritor `<applet>` e os parâmetros apropriados. Se a applet tiver parâmetros, em geral acompanha alguma informação de como usá-los.

Por exemplo, para incluir a applet `ImageTape` da Sun (vem junto com o JDK. Procure no diretório `java/demo/`) que faz com que um conjunto de imagens role na tela você precisa fazer o seguinte:

- Copie o arquivo `ImageTape.class` para o mesmo diretório onde está sua página HTML que irá chamá-lo.
- Renomeie as imagens da sua animação para `T1.gif`, `T2.gif`, `T3.gif`, etc. (isto é uma convenção do programador da applet). Vamos supor que você tem três imagens GIF de 150x85 pixels de dinossauros: `T1.gif`, `T2.gif` e `T3.gif`.

Crie um diretório onde ficarão as imagens (outra convenção do programador). Vamos chamá-lo de `sauros`.

- Escolha um lugar na sua página Web para colocá-lo, edite a página e acrescente o seguinte código:

```
<applet code="ImageTape.class" width=450 height=85>
  <param name=img value="sauros">
  <param name=nimgs value="3">
  <param name=speed value="10">
</applet>
```


O parâmetro `img` informa o nome do diretório onde estão as imagens `T1`, `T2`, etc. `nimgs` informa quantas são e `speed` diz a velocidade da animação. O tamanho da applet calculamos baseado na colocação das três imagens de 150 pixels lado a lado.

8.3. A classe Applet

O conjunto de classes, interfaces e métodos que compõem o pacote `java.applet` é pequeno. Consiste de uma única classe: Applet, e três interfaces: `AppletContext`, `AppletStub` e `AudioClip`. Applet é uma classe especializada da classe `Panel` que faz parte do pacote `java.awt`.

Applets são subclasses da classe `Applet`. Todo programa que roda como applet, tem na sua declaração "`extends Applet`":

```
public class Nome_da_Classe extends Applet
```

Após estender a classe applet, a nova subclasse deve sobrepor alguns de seus métodos. Nem todos precisam ser definidos explicitamente pelo programador. Em geral, o browser cuida da maioria dos detalhes.

Exemplos **[Java In A NutShell. 2nd. Ed.]**

Os dois exemplos a seguir ambos produzem um programa semelhante, onde se pode rabiscar.

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scribble2 extends Applet
                       implements MouseListener,  MouseMotionListener {
  private int last_x, last_y;

  public void init() {
    // Tell this applet what MouseListener and MouseMotionListener
    // objects to notify when mouse and mouse motion events occur.
```



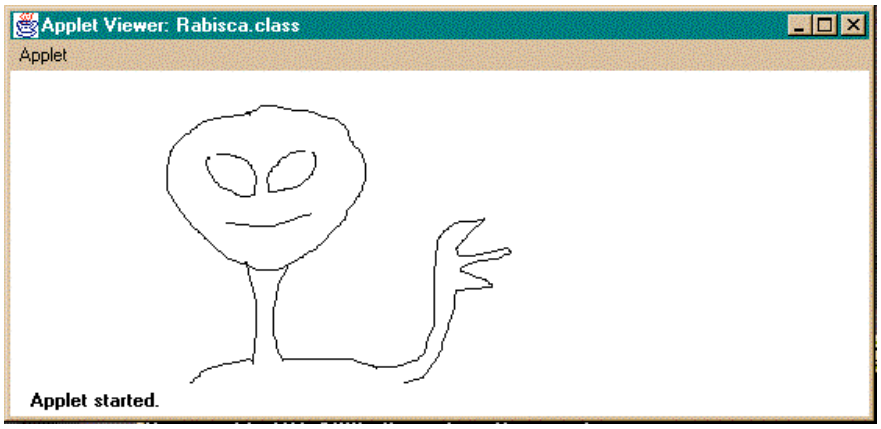
```
    // Since we implement the interfaces ourself, our own methods are called.
    this.addMouseListener(this);
    this.addMouseMotionListener(this);
  }

  // A method from the MouseListener interface.  Invoked when the
  // user presses a mouse button.
  public void mousePressed(MouseEvent e) {
    last_x = e.getX();
    last_y = e.getY();
  }

  // A method from the MouseMotionListener interface.  Invoked when the
  // user drags the mouse with a button pressed.
  public void mouseDragged(MouseEvent e) {
    Graphics g = this.getGraphics();
    int x = e.getX(), y = e.getY();
    g.drawLine(last_x, last_y, x, y);
    last_x = x; last_y = y;
  }

  // The other, unused methods of the MouseListener interface.
  public void mouseReleased(MouseEvent e) {;}
  public void mouseClicked(MouseEvent e) {;}
  public void mouseEntered(MouseEvent e) {;}
  public void mouseExited(MouseEvent e) {;}

  // The other method of the MouseMotionListener interface.
  public void mouseMoved(MouseEvent e) {;}
}
```

Este outro applet faz a mesma coisa que o exemplo anterior mas usando classes internas.[1]:

```
// This example is from the book "Java in a Nutshell, Second Edition".
// Written by David Flanagan.  Copyright (c) 1997 O'Reilly & Associates.
// You may distribute this source code for non-commercial purposes only.
// You may study, modify, and use this example for any purpose, as long as
// this notice is retained.  Note that this example is provided "as is",
// WITHOUT WARRANTY of any kind either expressed or implied.

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Scribble3 extends Applet {
  int last_x, last_y;

  public void init() {
    // Define, instantiate and register a MouseListener object.
    this.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
```



```
        last_x = e.getX();
        last_y = e.getY();
      }
    });

    // Define, instantiate and register a MouseMotionListener object.
    this.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        Graphics g = getGraphics();
        int x = e.getX(), y = e.getY();
        g.setColor(Color.black);
        g.drawLine(last_x, last_y, x, y);
        last_x = x; last_y = y;
      }
    });

    // Create a clear button
    Button b = new Button("Clear");
    // Define, instantiate, and register a listener to handle button presses
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {  // clear the scribble
        Graphics g = getGraphics();
        g.setColor(getBackground());
        g.fillRect(0, 0, getSize().width, getSize().height);
      }
    });
    // And add the button to the applet
    this.add(b);
  }
}
```



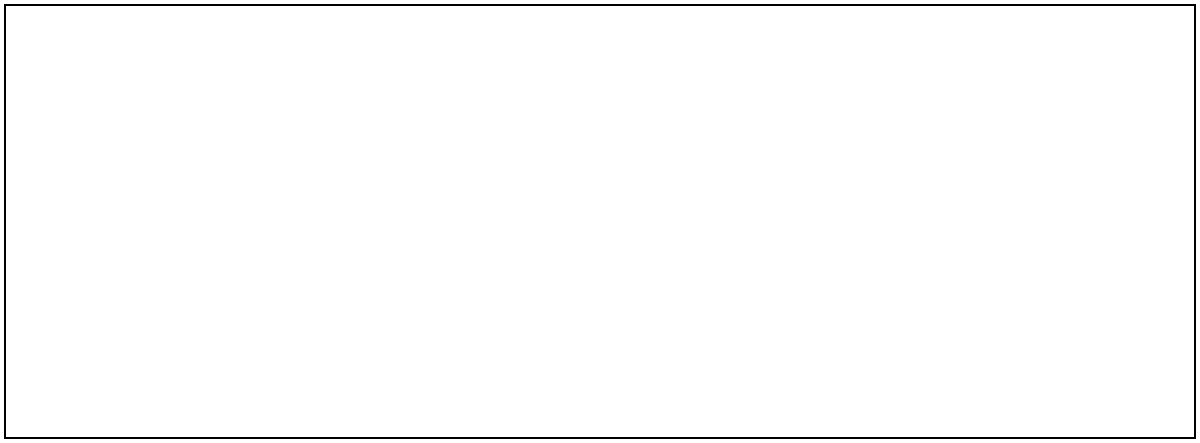
Métodos do ciclo de vida de um Applet

Os principais métodos de qualquer Applet são os que caracterizam o seu ciclo de vida. Applets rodam como `Threads`, e portanto, têm métodos parecidos com os de `Thread`. Os métodos abaixo são chamados automaticamente pelo browser. O programador do applet deve sobrepô-los se desejar que algum deles se comporte

de forma diferente (além desses, o método Component.`paint()` é chamado pelo sistema para desenhar o applet, ou qualquer um dos seus componentes).

- `public void init()`: este método realiza qualquer inicialização necessária na applet.
- `public void start()`: É o método que inicia a execução da applet. Geralmente é usado para iniciar um objeto `Thread` que roda o código principal da applet.
- `public void stop()`: Interrompe a execução da applet.
- `public void destroy()`: libera quaisquer recursos de memória que a applet esteja usando.

A figura abaixo ilustra o que ocorre durante o ciclo de vida de uma applet ao ser chamado pelo browser, visualizado e descartado. Qualquer um dos métodos pode ser sobreposto (de "Hooked on Java", Arthur Van Hoff, et Al, Addison Wesley, 1996).



Métodos de pintura

A pintura e re-pintura da tela são coisas que uma applet (ou qualquer aplicação AWT) típica faz várias vezes durante a sua execução. O método responsável por isso é o método `paint()`, também herdado por `java.awt.Component`. O sistema de execução Java se encarrega de chamar este método quando precisa. Nós, nos encarregamos de definir o que ele vai fazer.

A sua sintaxe básica é

```
public void paint(Graphics g)
```

`paint()` recebe como argumento um objeto do tipo `Graphics`, que é uma classe abstrata que define uma interface independente de plataforma para gráficos

(um contexto gráfico – área da tela do browser reservada para a execução do applet). Depois de criado um objeto `Graphics`, ele pode ser usado para alterar várias características de uma applet na tela. O contexto gráfico é fornecido pelo browser que define seu tamanho através de HTML. O método `paint()` o recebe e através dele pode desenhar na área do browser.

`paint()` é chamado automaticamente pelo sistema sempre que for necessário repintar a tela. Você nunca deve chamá-lo diretamente pois ele é chamado usando um *thread* diferente (o thread da AWT). Chama-lo no thread principal de sua aplicação poderá causar efeitos estranhos e atrapalhará a sua chamada normal, sempre que a tela for encoberta. O método `repaint()`, também disponível em qualquer componente, agenda uma chamada de `paint()` no thread do AWT através do método `update()`, que faz duas coisas: a) limpa a tela e; b) chama `paint()` para redesenhá-la. Os métodos `paint()` e `update()` rodam no mesmo thread (portanto não devem ser chamados diretamente). Se você quiser chamar `paint()` sem limpar a tela (se não quiser que a tela seja redesenhada), sobreponha `update()` da forma:

```
public void update(Graphics g) {
    paint(g);
}
```

Agora o método `repaint()` não mais limpará a tela antes de chamar `paint()`.

O seguinte exemplo (do "Java in a NutShell", mostra a utilização de `paint()` e objetos gráficos, para desenhar, mudar a cor e a fonte do texto.

```
// This example is from the book _Java in a Nutshell_ by David Flanagan.
// Written by David Flanagan.  Copyright (c) 1996 O'Reilly & Associates.
// You may study, use, modify, and distribute this example for any purpose.
// This example is provided WITHOUT WARRANTY either expressed or implied.

import java.applet.*;
import java.awt.*;


public class SecondApplet extends Applet {
    static final String message = "Java Colors";
    private Font font;

    // One-time initialization for the applet
    public void init() {
        font = new Font("Helvetica", Font.BOLD, 48);
```



```
    }

    // Draw the applet whenever necessary.  Do some fancy graphics.
    public void paint(Graphics g) {
        // The pink oval
        g.setColor(Color.pink);
        g.fillOval(10, 10, 330, 100);

        // The red outline.  java doesn't support wide lines, so we
        // try to simulate a 4-pixel wide line by drawing four ovals
        g.setColor(Color.red);
        g.drawOval(10,10, 330, 100);
        g.drawOval(9, 9, 332, 102);
        g.drawOval(8, 8, 334, 104);
        g.drawOval(7, 7, 336, 106);

        // The text
        g.setColor(Color.black);
        g.setFont(font);
        g.drawString(message, 40, 75);
    }
}
```

Métodos de Imagem e Som

O método

```
public Image getImage(URL url)
```

é usado pela `Applet` para recuperar uma imagem a partir de uma URL. Os métodos

```
public void play(URL url)
public AudioClip getAudioClip(URL url)
```

são usados para incluir e executar um clip sonoro em uma página através de uma applet. Para executar o clip uma única vez, usa-se `play()` com a URL do clip, formato "`*.au`" como argumento. Para fazer coisas mais sofisticadas, como fazer o clip reiniciar e repetir várias vezes, usa-se o método `getAudioClip()` associado a um objeto do tipo `AudioClip`, que pode então ser controlado de forma mais eficiente.

O exemplo a seguir usa imagens, `AudioClips` e eventos. É uma applet que mantém uma imagem apagada na tela até que o mouse esteja sobre a applet. Neste momento, a imagem "acende" e mostra suas cores originais. Se o mouse for clicado sobre a imagem, ele buscará uma URL definida por parâmetro:

```
/** Highlight Applet version 2 (uses Image Filters)
```



```
 */

import java.applet.*;
import java.net.*;
import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;


public class HighLight extends Applet implements Runnable, MouseListener {

    Image img, gray;
    URL url;
    String audio;
    Thread sndthread = null;

    public void init() {
        Dimension d = this.size();
        // busca URL da imagem;
        img = getImage(getDocumentBase(),getParameter("SRC"));

        // As três linhas a seguir criam uma cópia cinza da imagem
        ImageFilter f = new GrayFilter();
        ImageProducer producer = new FilteredImageSource(img.getSource(), f);
        gray = this.createImage(producer);

        // define tamanho da applet
        resize(d.width, d.height);

    }

    public void mouseReleased (MouseEvent e) {
        // quando mouse for liberado sobre Applet, acessa URL
        try {
            url = new URL(getDocumentBase(), getParameter("HREF"));
        } catch (MalformedURLException mue) {
            showStatus("Bad URL");
        }
        getAppletContext().showDocument(url);

    }

    public void mouseExited (MouseEvent e) {
        // quando mouse sair da applet, faça-o apagar
        repaint();
        if (sndthread == null) {
            sndthread = new Thread(this);
            sndthread.setPriority(Thread.MIN_PRIORITY);
            sndthread.start();
            audio = getParameter("OUTSND"); // arquivo de som (au)
            sndthread = null;               // opcional
        }
    }
```



```
    public void mouseEntered(MouseEvent e) {
        // when mowse enters applet, show image.
        Graphics g = this.getGraphics(); // new Graphics context
        g.drawImage(img, 0, 0, this);
        if (sndthread == null) {
            sndthread = new Thread(this);
            sndthread.setPriority(Thread.MIN_PRIORITY);
            sndthread.start();
            audio = getParameter("INSND");  // arquivo de som (au)
            sndthread = null;               // opcional
        }
    }

public void mouseClicked(MouseEvent e) {}
public void mousePressed(MouseEvent e) {}


    public void run() {
        play(getCodeBase(), audio);
    }

    public void paint(Graphics g) {
        // pinta imagem inicial de cinza
        g.drawImage(gray, 0, 0, this);
    }
}


// This class from "Java In a NutShell" by David Flanagan,
// O'Reilly and Associates. page 158
//
class GrayFilter extends RGBImageFilter {

    public GrayFilter() {
        canFilterIndexColorModel = true;
    }

    public int filterRGB(int x, int y, int rgb) {
        int a = rgb & 0xff000000;
        int r = ((rgb & 0xff0000) + 0xff0000)/2;
        int g = ((rgb & 0x00ff00) + 0x00ff00)/2;
        int b = ((rgb & 0x0000ff) + 0x0000ff)/2;
        return a | r | g | b;
    }
}
```

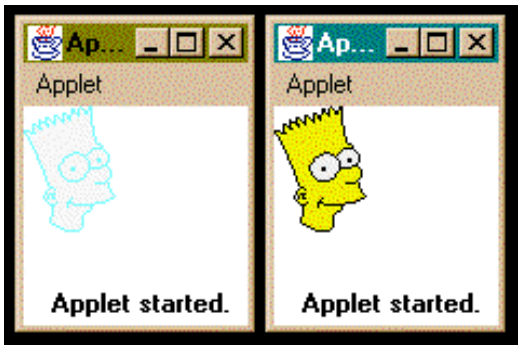
Os parâmetros usados (em `<param name="..." ...>`)são:

- `SRC` = URL da imagem a ser usada
- `HREF` = URL do link a ser acionado ao se clicar na imagem

e os opcionais:

- `INSND` = URL do arquivo de som que tocará quando o mouse entrar.
- `OUTSND` = URL do arquivo de som que tocará quando o mouse entrar.
- A classe `GrayFilter` implementa um método que "apaga" um pouco a imagem. Este método foi utilizado na applet para provocar o efeito apagado na imagem.

A figura abaixo ilustra o resultado (a segunda imagem aparece quando o mouse está sobre a applet):



Métodos para Leitura de Parâmetros HTML

Para ler parâmetros e outras informações a respeito dos arquivos HTML, a applet dispõe dos seguintes métodos:

- `public URL getDocumentBase()`: retorna a URL base do documento que contém a applet.
- `public URL getCodeBase()`: retorna a URL base da applet (do arquivo `.class`)
- `public String getParameter(String nome)`: retorna uma `String` contendo o nome de um parâmetro especificado no descritor `<param>` ou no descritor `<applet>`. Por exemplo, para ler os parâmetros da applet que faz rolar dinossauros na tela (que vimos no início deste capítulo), usaríamos:

```
String velocidade = getParameter("speed");
String imagens = getParameter("img");
String num_imagens = getParameter("nimgs");
```

8.4. Applets e componentes da AWT

Como a classe `Applet` herda toda a implementação da classe `java.awt.Component`, que é a principal classe do pacote `java.awt`, quase tudo o

que se pode fazer em relação à programação da GUI com o AWT, pode-se fazer em uma applet.

Uma exceção são os menus, que só podem ser incluídos em objetos da classe `java.awt.Frame`. `Frame` e `Panel` são duas subclasses descendentes de `Component`, mas `Panel` não descende de `Frame`, portanto Applet não herda a implementação de `Frame`.

No entanto, uma applet pode chamar um objeto `Frame` que abra como uma janela separada do browser, e nesta janela, colocar menus e tudo mais o que puder ser colocado em um `Frame`. Veja a especificação ou uma documentação específica sobre o AWT para maiores detalhes. Outra solução é usar JFC-Swing.

Exercício

1. Transforme uma das aplicações gráficas que você desenvolveu no capítulo anterior em uma Applet (não inclua os menus).
2. Escreva uma applet que desenhe um gráfico de barras a partir de informações passadas como parâmetros no HTML.

8.5. Gráficos

As outras classes do AWT que não abordamos são as que lidam com gráficos. São provavelmente as classes mais importantes já que definem os métodos de desenho de linhas, textos e pintura de imagens. Com elas é possível inclusive construir outros objetos (widgets) independentes de plataforma para a GUI (classe `java.awt.Toolkit`).

Neste curso, não entraremos nos detalhes de programação desta biblioteca gráfica, que certamente dariam um livro inteiro. Apenas apresentaremos a seguir os métodos construtores mais comuns das principais classes.

- **Color:** descreve uma cor. Esta classe contém constantes que definem cores. Para criar um objeto `Color`:

```
Color c = new Color(255, 0, 0);
```

- **Dimension:** um par de variáveis de instância que contém a largura e altura de alguma coisa. Uso típico em um applet, para capturar o seu tamanho:


```
Dimension d = this.size();
int largura = d.width;
int altura = d.height;
```

- **Font:** descreve uma fonte. Para criar um objeto `Font`:

```
Font f = new Font("Sans-Serif", Font.BOLD, 12);
```

- **Point:** armazena as coordenadas X e Y de um ponto bi-dimensional. O método `move()` define as coordenadas e `translate()` adiciona valores específicos às coordenadas. Vários métodos de outros objetos utilizam objetos do tipo `Point`.

```
Point p = new Point(pos_x, pos_y);
```

- **Polygon:** um polígono definido como uma matriz de pontos. Os pontos do polígono podem ser especificados com o método `addPoint(x, y)` ou através do construtor. O método `inside(x, y)` testa se um determinado ponto está contido no polígono.

```
Polygon py =
         new Polygon (x[], y[], num_pontos);
```

- **Rectangle:** define um retângulo pelas coordenadas do seu canto superior esquerdo e uma largura e altura, que podem ser manipulados livremente. Vários métodos de outros objetos utilizam objetos do tipo `Rectangle`. Uso típico:

```
Rectangle p = new Rectangle (x, y, larg, alt);
```

- **Graphics:** É uma classe abstrata que define vários métodos para desenho de linhas, de texto, pintura de imagens, cópia de áreas e recorte gráfico. Um objeto do tipo `Graphics` não pode ser criado diretamente. Precisa ser obtido via um método `getGraphics()` de um `Component` ou `Image`. O uso mais comum é como argumento do método `paint()` de um `Component`:

```
public void paint(Graphics g);
```

  O Java 2 (JDK1.2) aumentou bastante os recursos de `Graphics` na classe `Graphics2D`. Para usar `Graphics2D` basta fazer uma conversão, dentro do método paint, antes do uso:

```
public void paint(Graphics g) {
    Graphics g2 = (Graphics2D) g;
```



```
        g2.setColor(...) ; métodos de Graphics2D
    }
```

- **Image:** é uma classe abstrata que representa uma imagem de uma forma independente de plataforma. Deve ser obtida através de uma chamada `Applet.getImage()` ou `Component.createImage()`. Os métodos mais importantes são: `getGraphics()`, que retorna um objeto gráfico que pode ser usado para desenhar em imagens na memória; e `getSource()`, que retorna um objeto `ImageProducer` que gera a imagem.

  O trecho de código a seguir é usado para pintar uma imagem na memória (usando um objeto `Image`, para armazená-lo). A vantagem de se pintar uma imagem na memória antes de se utilizá-la, está em evitar piscadas em animações, causadas pelo redesenho da tela. Esta técnica se chama "double-buffering":

```
(...)
Dimension d = this.size(); // tam. da applet

// cria objeto Image offscreen (na memória)
Image offscreen = this.createImage(d.width, d.height);

// cria contexto grafico em offscreen
Graphics g = offscreen.getGraphics();

// chama paint() e pinta imagem em offscreen
paint(g);

// cria contexto grafico em Applet (na tela)
g = this.getGraphics();


// desenha imagem em Applet (aparece na tela)
g.drawImage(offscreen, 0, 0, this);
```


NÃO É JAVA

9

Hora do intervalo...

Lima Barreto

O Homem Que Sabia Javanês

EM UMA confeitaria, certa vez, ao meu amigo Castro, contava eu as partidas que havia pregado às convicções e às respeitabilidades, para poder viver.

Houve mesmo, uma dada ocasião, quando estive em Manaus, em que fui obrigado a esconder a minha qualidade de bacharel, para mais confiança obter dos clientes, que afluíam ao meu escritório de feiticeiro e adivinho. Contava eu isso.

O meu amigo ouvia-me calado, embevecido, gostando daquele meu Gil Blas vivido, até que, em uma pausa da conversa, ao esgotarmos os copos, observou a esmo:

— Tens levado uma vida bem engraçada, Castelo !

— Só assim se pode viver... Isto de uma ocupação única: sair de casa a certas horas, voltar a outras, aborrece, não achas? Não sei como me tenho agüentado lá, no consulado !

— Cansa-se; mas, não é disso que me admiro. O que me admira, é que tenhas corrido tantas aventuras aqui, neste Brasil imbecil e burocrático.

— Qual! Aqui mesmo, meu caro Castro, se podem arranjar belas páginas de vida. Imagina tu que eu já fui professor de javanês!

— Quando? Aqui, depois que voltaste do consulado?

— Não; antes. E, por sinal, fui nomeado cônsul por isso.

— Conta lá como foi. Bebes mais cerveja?

— Bebo.

Mandamos buscar mais outra garrafa, enchemos os copos, e continuei:

— Eu tinha chegado havia pouco ao Rio estava literalmente na miséria. Vivia fugido de casa de pensão em casa de pensão, sem saber onde e como ganhar dinheiro, quando li no Jornal do Comércio o anuncio seguinte:

"Precisa-se de um professor de língua javanesa. Cartas, etc." Ora, disse cá comigo, está ali uma colocação que não terá muitos concorrentes; se eu capiscasse quatro palavras, ia apresentar-me. Saí do café e andei pelas ruas, sempre a imaginar-me professor de javanês, ganhando dinheiro, andando de bonde e sem encontros desagradáveis com os "cadáveres". Insensivelmente dirigi-me à Biblioteca Nacional. Não sabia bem que livro iria pedir; mas, entrei, entreguei o chapéu ao porteiro, recebi a senha e subi. Na escada, acudiu-me pedir a *Grande Encyclopédie*, letra J, a fim de consultar o artigo relativo a Java e a língua javanesa. Dito e feito. Fiquei sabendo, ao fim de alguns minutos, que Java era uma grande ilha do arquipélago de Sonda, colônia holandesa, e o javanês, língua aglutinante do grupo maleo-polinésico, possuía uma literatura digna de nota e escrita em caracteres derivados do velho alfabeto hindu.

A *Encyclopédie* dava-me indicação de trabalhos sobre a tal língua malaia e não tive dúvidas em consultar um deles. Copiei o alfabeto, a sua pronunciação figurada e saí. Andei pelas ruas, perambulando e mastigando letras. Na minha cabeça dançavam hieróglifos; de quando em quando consultava as minhas notas; entrava nos jardins e escrevia estes calungas na areia para guardá-los bem na memória e habituar a mão a escrevê-los.

À noite, quando pude entrar em casa sem ser visto, para evitar indiscretas perguntas do encarregado, ainda continuei no quarto a engolir o meu "a-b-c" malaio, e, com tanto afinco levei o propósito que, de manhã, o sabia perfeitamente.

Convenci-me que aquela era a língua mais fácil do mundo e saí; mas não tão cedo que não me encontrasse com o encarregado dos aluguéis dos cômodos:

— Senhor Castelo, quando salda a sua conta?

Respondi-lhe então eu, com a mais encantadora esperança:

— Breve... Espere um pouco... Tenha paciência... Vou ser nomeado professor de javanês, e...

Por aí o homem interrompeu-me:

— Que diabo vem a ser isso, Senhor Castelo?

Gostei da diversão e ataquei o patriotismo do homem:

— É uma língua que se fala lá pelas bandas do Timor. Sabe onde é?

Oh! alma ingênua! O homem esqueceu-se da minha dívida e disse-me com aquele falar forte dos portugueses:

— Eu cá por mim, não sei bem; mas ouvi dizer que são umas terras que temos lá para os lados de Macau. E o senhor sabe isso, Senhor Castelo?

Animado com esta saída feliz que me deu o javanês, voltei a procurar o anúncio. Lá estava ele. Resolvi animosamente propor-me ao professorado do idioma oceânico. Redigi a resposta, passei pelo Jornal e lá deixei a carta. Em seguida, voltei à biblioteca e continuei os meus estudos de javanês. Não fiz grandes progressos nesse dia, não sei se por julgar o alfabeto javanês o único saber necessário a um professor de língua malaia ou se por ter me empenhado mais na bibliografia e história literária do idioma que ia ensinar.

Ao cabo de dois dias, recebia eu uma carta para ir falar ao doutor Manuel Feliciano Soares Albernaz, Barão de Jacuecanga, à Rua Conde de Bonfim, não me recordo bem que numero. E preciso não te esqueceres que entrementes continuei estudando o meu malaio, isto é, o tal javanês. Além do alfabeto, fiquei sabendo o nome de alguns autores, também perguntar e responder "como está o senhor?" — e duas ou três regras de gramática, lastrado todo esse saber com vinte palavras do léxico.

Não imaginas as grandes dificuldades com que lutei, para arranjar os quatrocentos réis da viagem! É mais fácil — podes ficar certo — aprender o javanês... Fui a pé. Cheguei suadíssimo; e, Com maternal carinho, as anosas mangueiras, que se perfilavam em alameda diante da casa do titular, me receberam, me acolheram e me reconfortaram. Em toda a minha vida, foi o único momento em que cheguei a sentir a simpatia da natureza...

Era uma casa enorme que parecia estar deserta; estava mal tratada, mas não sei porque me veio pensar que nesse mau tratamento havia mais desleixo e cansaço de viver que mesmo pobreza. Devia haver anos que não era pintada. As paredes descascavam e os beirais do telhado, daquelas telhas vidradas de outros tempos, estavam desguarnecidos aqui e ali, como dentaduras decadentes ou mal cuidadas.

Olhei um pouco o jardim e vi a pujança vingativa com que a tiririca e o carrapicho tinham expulsado os tinhorões e as begônias. Os crótons continuavam, porém, a viver com a sua folhagem de cores mortiças. Bati. Custaram-me a abrir. Veio, por fim, um antigo preto africano, cujas barbas e cabelo de algodão davam à sua fisionomia uma aguda impressão de velhice, doçura e sofrimento.

Na sala, havia uma galeria de retratos: arrogantes senhores de barba em colar se perfilavam enquadrados em imensas molduras douradas, e doces perfis de senhoras, em bandós, com grandes leques, pareciam querer subir aos ares, enfunadas pelos redondos vestidos à balão; mas, daquelas velhas coisas, sobre as quais a poeira punha mais antiguidade e respeito, a que gostei mais de ver foi um belo jarrão de porcelana da China ou da Índia, como se diz. Aquela pureza da louça, a sua fragilidade, a ingenuidade do desenho e aquele seu fosco brilho de luar, diziam-me a mim que aquele objeto tinha sido feito por mãos de criança, a sonhar, para encanto dos olhos fatigados dos velhos desiludidos...

Esperei um instante o dono da casa. Tardou um pouco. Um tanto trôpego, com o lenço de alcobaça na mão, tomando veneravelmente o simonte de antanho, foi cheio de respeito que o vi chegar. Tive vontade de ir-me embora. Mesmo se não fosse ele o discípulo, era sempre um crime mistificar aquele ancião, cuja velhice trazia à tona do meu pensamento alguma coisa de augusto, de sagrado. Hesitei, mas fiquei.

— Eu sou, avancei, o professor de javanês, que o senhor disse precisar.

— Sente-se, respondeu-me o velho. O senhor é daqui, do Rio?

— Não, sou de Canavieiras.

— Como? fez ele. Fale um pouco alto, que sou surdo, — Sou de Canavieiras, na Bahia, insisti eu. — Onde fez os seus estudos?

— Em São Salvador.

— E onde aprendeu o javanês? indagou ele, com aquela teimosia peculiar aos velhos.

Não contava com essa pergunta, mas imediatamente arquitetei uma mentira. Contei-lhe que meu pai era javanês. Tripulante de um navio mercante, viera ter à Bahia, estabelecera-se nas proximidades de Canavieiras como pescador, casara, prosperara e fora com ele que aprendi javanês.

— E ele acreditou? E o físico? perguntou meu amigo, que até então me ouvira calado.

— Não sou, objetei, lá muito diferente de um javanês. Estes meus cabelos corridos, duros e grossos e a minha pele basané podem dar-me muito bem o aspecto de um mestiço de malaio...Tu sabes bem que, entre nós, há de tudo: índios, malaios, taitianos, malgaches, guanches, até godos. É uma comparsaria de raças e tipos de fazer inveja ao mundo inteiro.

— Bem, fez o meu amigo, continua.

— O velho, emendei eu, ouviu-me atentamente, considerou demoradamente o meu físico, pareceu que me julgava de fato filho de malaio e perguntou-me com doçura:

— Então está disposto a ensinar-me javanês?

— A resposta saiu-me sem querer: — Pois não.

— O senhor há de ficar admirado, aduziu o Barão de Jacuecanga, que eu, nesta idade, ainda queira aprender qualquer coisa, mas...

— Não tenho que admirar. Têm-se visto exemplos e exemplos muito fecundos... ? .

— O que eu quero, meu caro senhor....

— Castelo, adiantei eu.

— O que eu quero, meu caro Senhor Castelo, é cumprir um juramento de família. Não sei se o senhor sabe que eu sou neto do Conselheiro Albernaz, aquele que acompanhou Pedro I, quando abdicou. Voltando de Londres, trouxe para aqui um livro em língua esquisita, a que tinha grande estimação. Fora um hindu ou siamês que lho dera, em Londres, em agradecimento a não sei que serviço prestado por meu avô. Ao morrer meu avô, chamou meu pai e lhe disse: "Filho, tenho este livro aqui, escrito em javanês. Disse-me quem mo deu que ele evita desgraças e traz felicidades para quem o tem. Eu não sei nada ao certo. Em todo o caso, guarda-o; mas, se queres que o fado que me deitou o sábio oriental se cumpra, faze com que teu filho o entenda, para que sempre a nossa raça seja feliz." Meu pai, continuou o velho barão, não acreditou muito na história; contudo, guardou o livro. Às portas da morte, ele mo deu e disse-me o que prometera ao pai. Em começo, pouco caso fiz da história do livro. Deitei-o a um canto e fabriquei minha vida. Cheguei até a esquecer-me dele; mas, de uns tempos a esta parte, tenho passado por tanto desgosto, tantas desgraças têm caído sobre a minha velhice que me 1embrei do talismã da família. Tenho que o ler, que o compreender, se não quero que os meus últimos dias anunciem o desastre da minha posteridade; e, para entendê-lo, é claro, que preciso entender o javanês. Eis aí.

Calou-se e notei que os olhos do velho se tinham orvalhado. Enxugou discretamente os olhos e perguntou-me se queria ver o tal livro. Respondi-lhe que sim. Chamou o criado, deu-lhe as instruções e explicou-me que perdera todos os filhos, sobrinhos, só lhe restando uma filha casada, cuja prole, porém, estava reduzida a um filho, débil de corpo e de saúde frágil e oscilante.

Veio o livro. Era um velho calhamaço, um *in-quarto* antigo, encadernado em couro, impresso em grandes letras, em um papel amarelado e grosso. Faltava a folha do rosto e por isso não se podia ler a data da impressão. Tinha ainda umas páginas de prefácio, escritas em inglês, onde li que se tratava das histórias do príncipe Kulanga, escritor javanês de muito mérito.

Logo informei disso o velho barão que, não percebendo que eu tinha chegado aí pelo inglês, ficou tendo em alta consideração o meu saber malaio. Estive ainda folheando o cartapácio, à laia de quem sabe magistralmente aquela espécie de vasconço, até que afinal contratamos as condições de preço e de hora, comprometendo-me a fazer com que ele lesse o tal alfarrábio antes de um ano.

Dentro em pouco, dava a minha primeira lição, mas o velho não foi tão diligente quanto eu. Não conseguia aprender a distinguir e a escrever nem sequer quatro letras. Enfim, com metade do alfabeto levamos um mês e o Senhor Barão de Jacuecanga não ficou lá muito senhor da matéria: aprendia e desaprendia.

A filha e o genro (penso que até aí nada sabiam da história do livro) vieram a ter notícias do estudo do velho; não se incomodaram. Acharam graça e julgaram a coisa boa para distraí-lo.

Mas com o que tu vais ficar assombrado, meu caro Castro, é com a admiração que o genro ficou tendo pelo professor de javanês. Que coisa Única! Ele não se cansava de repetir: "É um assombro! Tão moço! Se eu soubesse isso, ah! onde estava !"

O marido de Dona Maria da Glória (assim se chamava a filha do barão), era desembargador, homem relacionado e poderoso; mas não se pejava em mostrar diante de todo o mundo a sua admiração pelo meu javanês. Por outro lado, o barão estava contentíssimo. Ao fim de dois meses, desistira da aprendizagem e pedira-me que lhe traduzisse, um dia sim outro não, um trecho do livro encantado. Bastava entendê-lo, disse-me ele; nada se opunha que outrem o traduzisse e ele ouvisse. Assim evitava a fadiga do estudo e cumpria o encargo.

Sabes bem que até hoje nada sei de javanês, mas compus umas histórias bem tolas e impingi-as ao velhote como sendo do crônicon. Como ele ouvia aquelas bobagens !...

Ficava extático, como se estivesse a ouvir palavras de um anjo. E eu crescia aos seus olhos !

Fez-me morar em sua casa, enchia-me de presentes, aumentava-me o ordenado. Passava, enfim, uma vida regalada.

Contribuiu muito para isso o fato de vir ele a receber uma herança de um seu parente esquecido que vivia em Portugal. O bom velho atribuiu a cousa ao meu javanês; e eu estive quase a crê-lo também.

Fui perdendo os remorsos; mas, em todo o caso, sempre tive medo que me aparecesse pela frente alguém que soubesse o tal patuá malaio. E esse meu temor foi grande, quando o doce barão me mandou com uma carta ao Visconde de Caruru, para que me fizesse entrar na diplomacia. Fiz-lhe todas as objeções: a minha fealdade, a falta de elegância, o meu aspecto tagalo. — "Qual! retrucava ele. Vá, menino; você sabe javanês!" Fui. Mandou-me o visconde para a Secretaria dos Estrangeiros com diversas recomendações. Foi um sucesso.

O diretor chamou os chefes de secção: "Vejam só, um homem que sabe javanês — que portento!"

Os chefes de secção levaram-me aos oficiais e amanuenses e houve um destes que me olhou mais com ódio do que com inveja ou admiração. E todos diziam: "Então sabe javanês? É difícil? Não há quem o saiba aqui!"

O tal amanuense, que me olhou com ódio, acudiu então: "É verdade, mas eu sei canaque. O senhor sabe?" Disse-lhe que não e fui à presença do ministro.

A alta autoridade levantou-se, pôs as mãos às cadeiras, concertou o *pince-nez* no nariz e perguntou: "Então, sabe javanês?" Respondi-lhe que sim; e, à sua pergunta onde o tinha aprendido, contei-lhe a história do tal pai javanês. "Bem, disse-me o ministro, o senhor não deve ir para a diplomacia; o seu físico não se presta... O bom seria um consulado na Ásia ou Oceania. Por ora, não há vaga, mas vou fazer uma reforma e o senhor entrará. De hoje em diante, porém, fica adido ao meu ministério e quero que, para o ano, parta para Bâle, onde vai representar o Brasil no Congresso de Lingüística. Estude, leia o Hovelacque, o Max Müller, e outros!"

Imagina tu que eu até aí nada sabia de javanês, mas estava empregado e iria representar o Brasil em um congresso de sábios.

O velho barão veio a morrer, passou o livro ao genro para que o fizesse chegar ao neto, quando tivesse a idade conveniente e fez-me uma deixa no testamento.

Pus-me com afã no estudo das línguas maleo-polinésicas; mas não havia meio!

Bem jantado, bem vestido, bem dormido, não tinha energia necessária para fazer entrar na cachola aquelas coisas esquisitas. Comprei livros, assinei revistas: *Revue Anthropologique et Linguistique, Proceedings of the English-Oceanic Association, Archivo Glottologico Italiano*, o diabo, mas nada! E a minha fama crescia. Na rua, os informados apontavam-me, dizendo aos outros: "Lá vai o sujeito que sabe javanês." Nas livrarias, os gramáticos consultavam-me sobre a colocação dos pronomes no tal jargão das ilhas de Sonda. Recebia cartas dos eruditos do interior, os jornais citavam o meu saber e recusei aceitar uma turma de alunos sequiosos de entenderem o tal javanês. A convite da redação, escrevi, no *Jornal do Comércio* um artigo de quatro colunas sobre a literatura javanesa antiga e moderna...

— Como, se tu nada sabias? interrompeu-me o atento Castro.

— Muito simplesmente: primeiramente, descrevi a ilha de Java, com o auxílio de dicionários e umas poucas de geografias, e depois citei a mais não poder.

— E nunca duvidaram? perguntou-me ainda o meu amigo.

— Nunca. Isto é, uma vez quase fico perdido. A polícia prendeu um sujeito, um marujo, um tipo bronzeado que só falava uma língua esquisita. Chamaram diversos intérpretes, ninguém o entendia. Fui também chamado, com todos os respeitos que a minha sabedoria merecia, naturalmente. Demorei-me em ir, mas fui afinal. O homem já estava solto, graças à intervenção do cônsul holandês, a quem ele se fez compreender com meia dúzia de palavras holandesas. E o tal marujo era javanês — uf!

Chegou, enfim, a época do congresso, e lá fui para a Europa. Que delícia! Assisti à inauguração e às sessões preparatórias. Inscreveram-me na secção do tupi-guarani e eu abalei para Paris. Antes, porém, fiz publicar no *Mensageiro de Bâle* o meu retrato, notas biográficas e bibliográficas. Quando voltei, o presidente pediu-me desculpas por me ter dado aquela secção; não conhecia os meus trabalhos e julgara que, por ser eu americano brasileiro, me estava naturalmente indicada a secção do tupi— guarani. Aceitei as explicações e até hoje ainda não pude escrever as minhas obras sobre o javanês, para lhe mandar, conforme prometi.

Acabado o congresso, fiz publicar extratos do artigo do *Mensageiro de Bâle*, em Berlim, em Turim e Paris, onde os leitores de minhas obras me ofereceram um banquete, presidido pelo Senador Gorot. Custou-me toda essa brincadeira, inclusive o banquete que me foi oferecido, cerca de dez mil francos, quase toda a herança do crédulo e bom Barão de Jacuecanga.

Não perdi meu tempo nem meu dinheiro. Passei a ser uma glória nacional e, ao saltar no cais Pharoux, recebi uma ovação de todas as classes sociais e o presidente da república, dias depois, convidava-me para almoçar em sua companhia.

Dentro de seis meses fui despachado cônsul em Havana, onde estive seis anos e para onde voltarei, a fim de aperfeiçoar os meus estudos das línguas da Malaia, Melanésia e Polinésia.

— É fantástico, observou Castro, agarrando o copo de cerveja.

— Olha: se não fosse estar contente, sabes que ia ser ?

— Que?

— Bacteriologista eminente. V amos?

— Vamos.

Gazeta da Tarde, Rio.28-4-1911.

O homem que sabia Javanês

Fonte:

BARRETO, Lima. *O homem que sabia javanês e outros contos*. Curitiba: Polo Editorial do Paraná, 1997.

Texto proveniente de:
A Biblioteca Virtual do Estudante Brasileiro
<http://www.bibvirt.futuro.usp.br>
A Escola do Futuro da Universidade de São Paulo
Permitido o uso apenas para fins educacionais.
Texto-base digitalizado por:
Rodrigo Souza, Curitiba — PR

JAVA 10

Uma aplicação distribuída

A FINALIDADE DESTE MÓDULO É APRESENTAR UM EXEMPLO DIDÁTICO que ilustre o uso prático das tecnologias discutidas nos módulos anteriores (e no módulo de servlets) em cenários onde aplicações distribuídas são necessárias. O problema de acesso a um banco de dados localizado em uma rede TCP/IP será solucionado empregando tecnologias como CORBA, RMI e soquetes TCP em aplicações executando em três cenários diferentes. Os cenários se distinguem pela interface do usuário e plataforma utilizada. Na plataforma Web teremos um applet Java rodando no browser, e páginas HTML geradas por um servlet rodando no servidor HTTP. Na plataforma *Windows* teremos um programa executável Java. Este módulo serve de referência e é essencial para a total compreensão dos exemplos dos módulos Java 6 e Java 7.

Tópicos abordados neste módulo

- Execução e instalação de uma aplicação Java multiplataforma cujo código foi analisado em módulos anteriores
- Discussão a respeito de desempenho e aplicabilidade de cada solução em diversos cenários.

Índice

Objetivos

No final deste módulo você deverá ser capaz de:

- comparar as várias soluções de conectividade usando Java e saber escolher a mais adequada a determinado ambiente.
- perceber as vantagens do uso de interfaces e outros mecanismos orientados a objetos que promovem o reuso de componentes.

10.1.Introdução

A aplicação analisada permite o acesso a um banco de informações armazenadas em um arquivo de texto ou em banco de dados relacional, localizado em um servidor remoto. O acesso pode ser direto ou através de uma das quatro aplicações intermediárias que interceptam as requisições do cliente. Essas aplicações foram construídas para atuar como servidores e realizar a comunicação usando CORBA, RMI, RMI sobre IIOP ou soquetes TCP (`java.net`).

As aplicações mencionadas acima executam sob o sistema operacional local. Também analisaremos aplicações que funcionam sob a plataforma Web. São mais duas versões da aplicação de banco de dados, usando tecnologias Web *client-side* e *server-side*. A primeira versão, consiste de uma interface do usuário proporcionada por um applet Java – cuja lógica da aplicação reside no browser. A segunda aplicação Web utiliza HTML e JavaScript como interface do usuário e concentra a lógica da aplicação em um servlet Java instalado no servidor.

10.2. Apresentação e execução das aplicações

As figuras 10-1 a 10-5 ilustram as interfaces do usuário das aplicações analisadas. Todas possuem o mesmo núcleo. Foram construídas separando a lógica da aplicação da interface do usuário e de armazenamento.

Aplicações independentes (executam sob o sistema operacional)

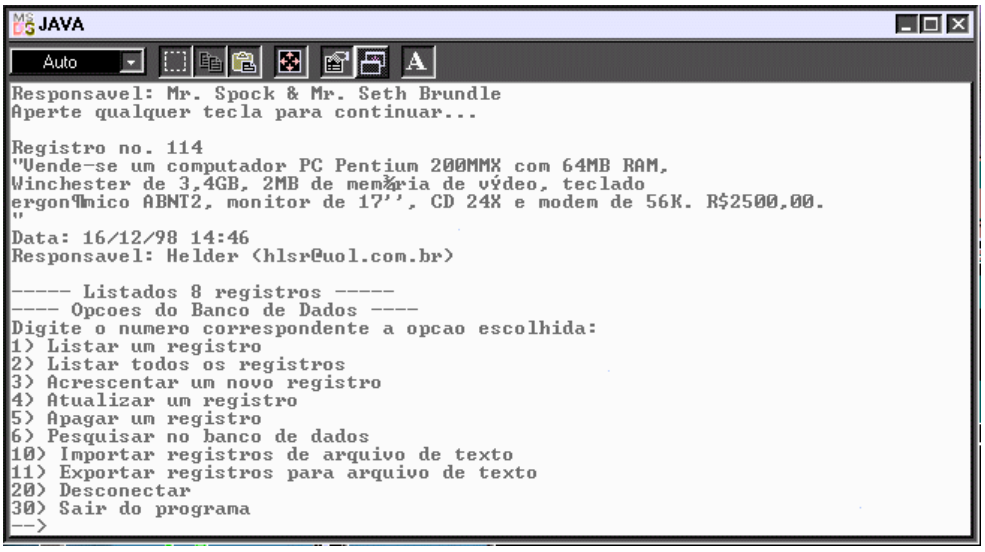


Figura 10-1 - Aplicação cliente com interface do usuário orientada a caracter

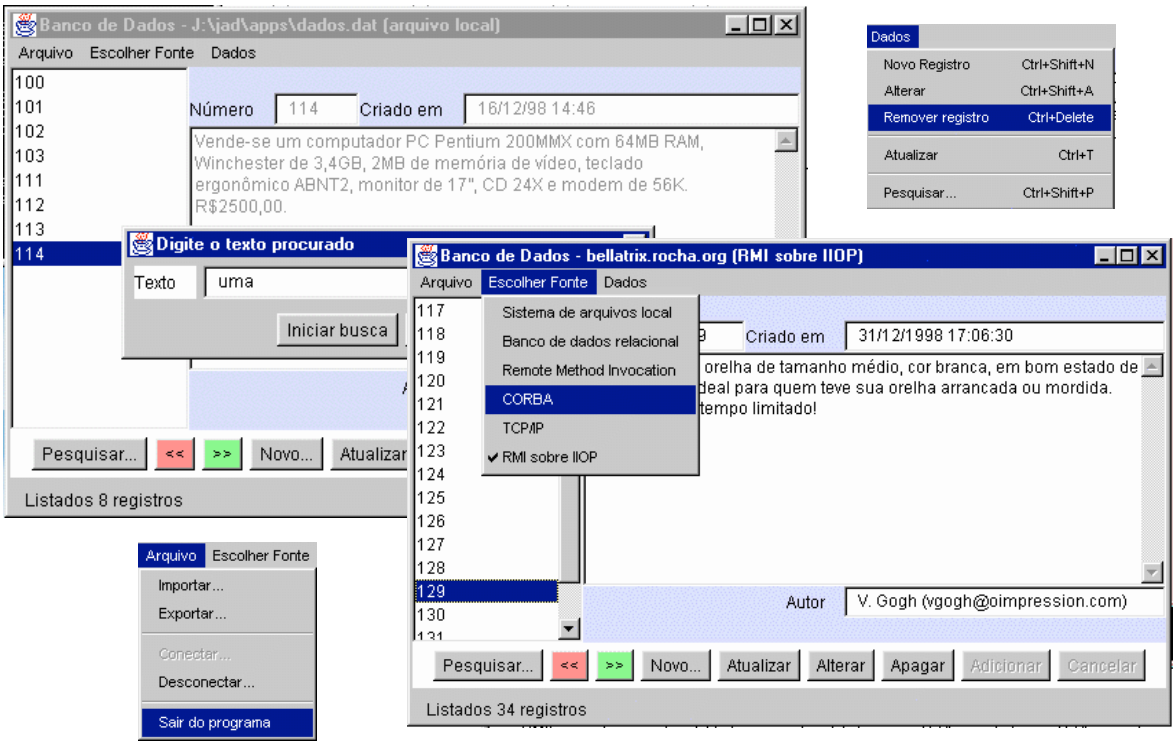


Figura 10-2 – Aplicação cliente com interface gráfica. (a) durante opções "Pesquisar..." em arquivo local; (b) e (c) menus; (d) listando os dados de banco de dados remoto (via RMI/IIOP)

Aplicações Web: Applets e Servlets

Figura 10-3 (a)– Interface cliente como applet em browser Netscape(com opção "pesquisar..." ativada)

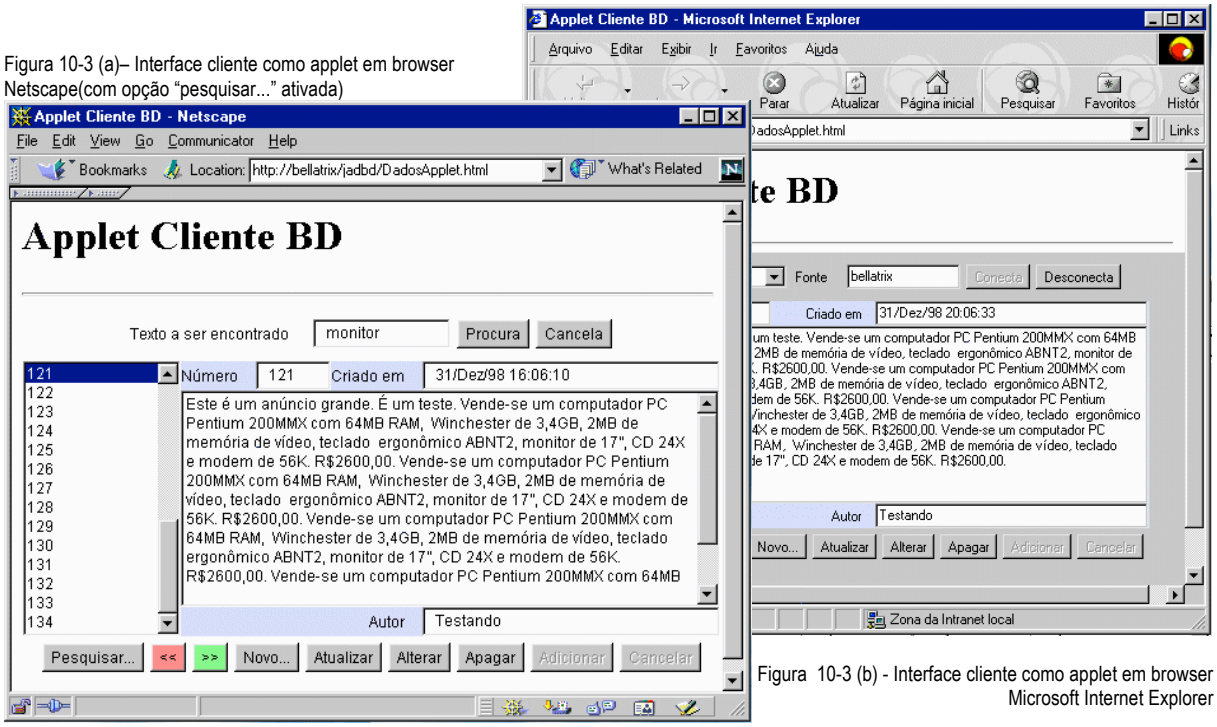


Figura 10-3 (b) - Interface cliente como applet em browser Microsoft Internet Explorer

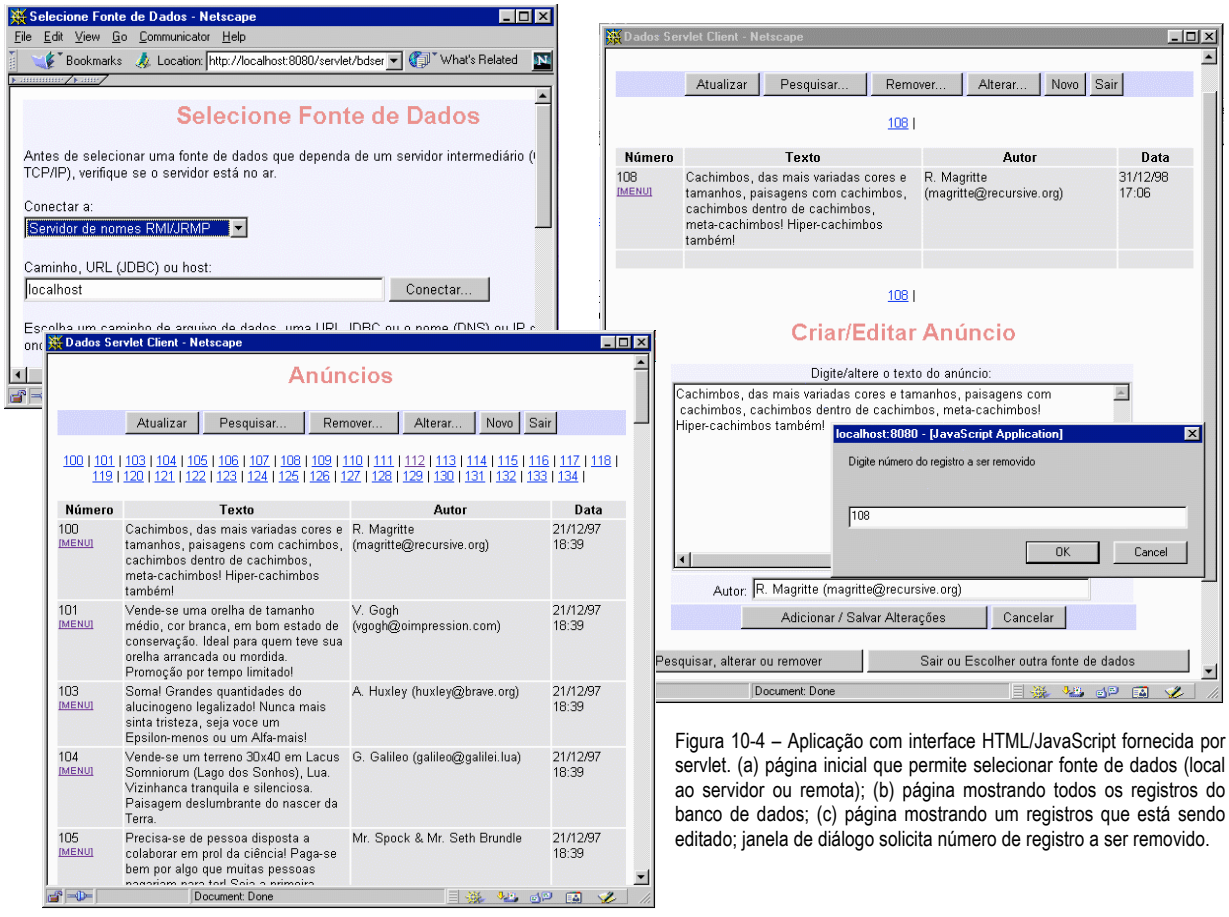


Figura 10-4 – Aplicação com interface HTML/JavaScript fornecida por servlet. (a) página inicial que permite selecionar fonte de dados (local ao servidor ou remota); (b) página mostrando todos os registros do banco de dados; (c) página mostrando um registros que está sendo editado; janela de diálogo solicita número de registro a ser removido.

Aplicações intermediárias (servidores)

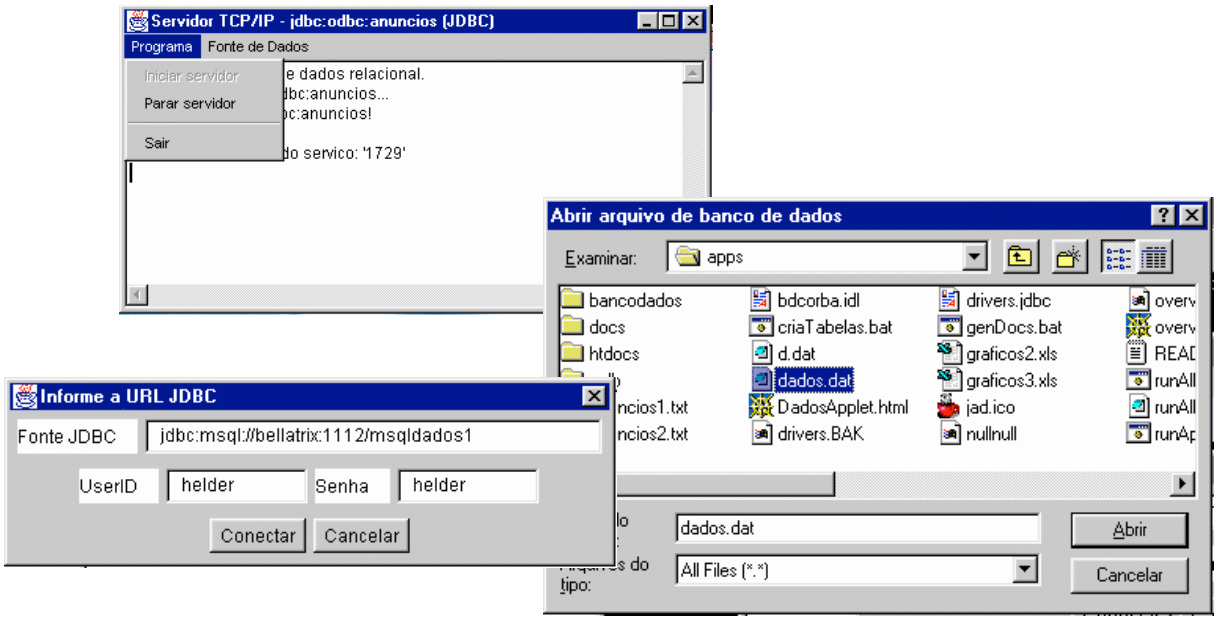


Figura 10-5 – (a) Aparência geral dos servidores intermediários (BDProtocol, CORBA, RMI, RMI/IIOP).
(b) Diálogo para conectar com banco de dados em arquivo. (c) Diálogo para conectar em banco de dados relacional através de URL JDBC

O banco de dados consiste de um conjunto de "anúncios" com número, nome, data e conteúdo. Todas as interfaces do usuário são adaptadas para entender esse formato. A tecnologia utilizada para armazenamento e a tecnologia utilizada para a comunicação remota podem ser diferentes. Todas são também suportadas pela aplicação. Em todas as interfaces console (figura 10-1), gráfica (10-2), applet (10-3) e servlet (10-4) é possível escolher se a transferência das informações será através de acesso local ou remoto usando CORBA, RMI, RMI sobre IIOP e TCP/IP. A aplicação também suporta qualquer banco de dados JDBC ou um formato proprietário baseado em arquivo para armazenar os dados. Os vários "blocos destacáveis" da aplicação estão ilustrados na figura 10-6.

Não é objetivo deste capítulo comparar o desempenho entre tecnologias de objetos remotos, portanto, as diferenças entre CORBA, RMI e TCP/IP não serão consideradas nas aplicações analisadas aqui. Pretendemos, porém, discutir o funcionamento de uma aplicação funcionando com interface applet, HTML com servlet ou como executável do sistema operacional. Para isto, mediremos tempos de resposta e requisição entre cliente e servidor usando uma das tecnologias de rede (TCP/IP), e cada uma das três interfaces. As configurações utilizadas nos experimentos deste capitulo estão ilustradas na figura 10-7.

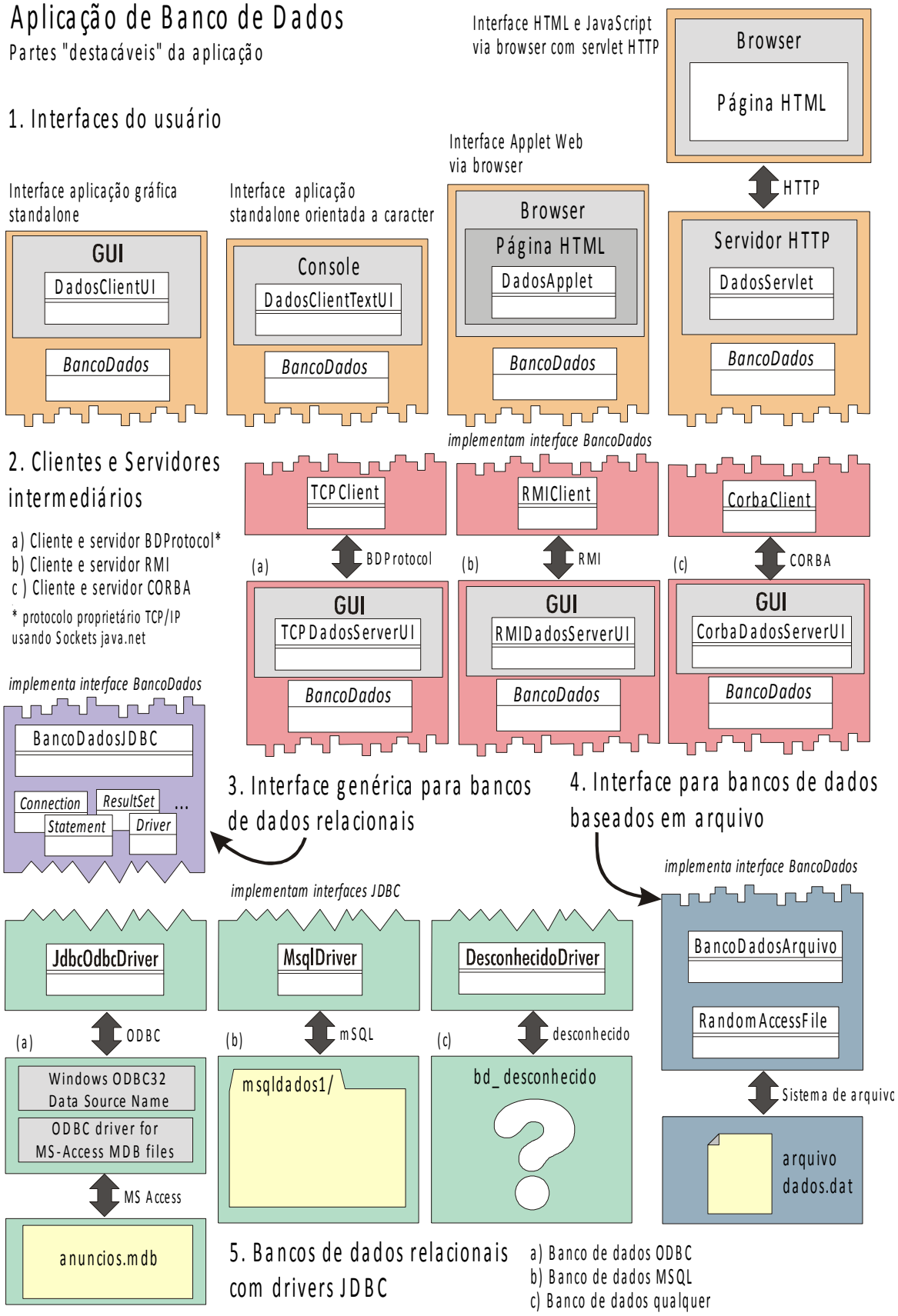


Figura 10-6 - Partes destacáveis da aplicação "bancodados". Veja mais detalhes sobre a estrutura da aplicação no apêndice C.

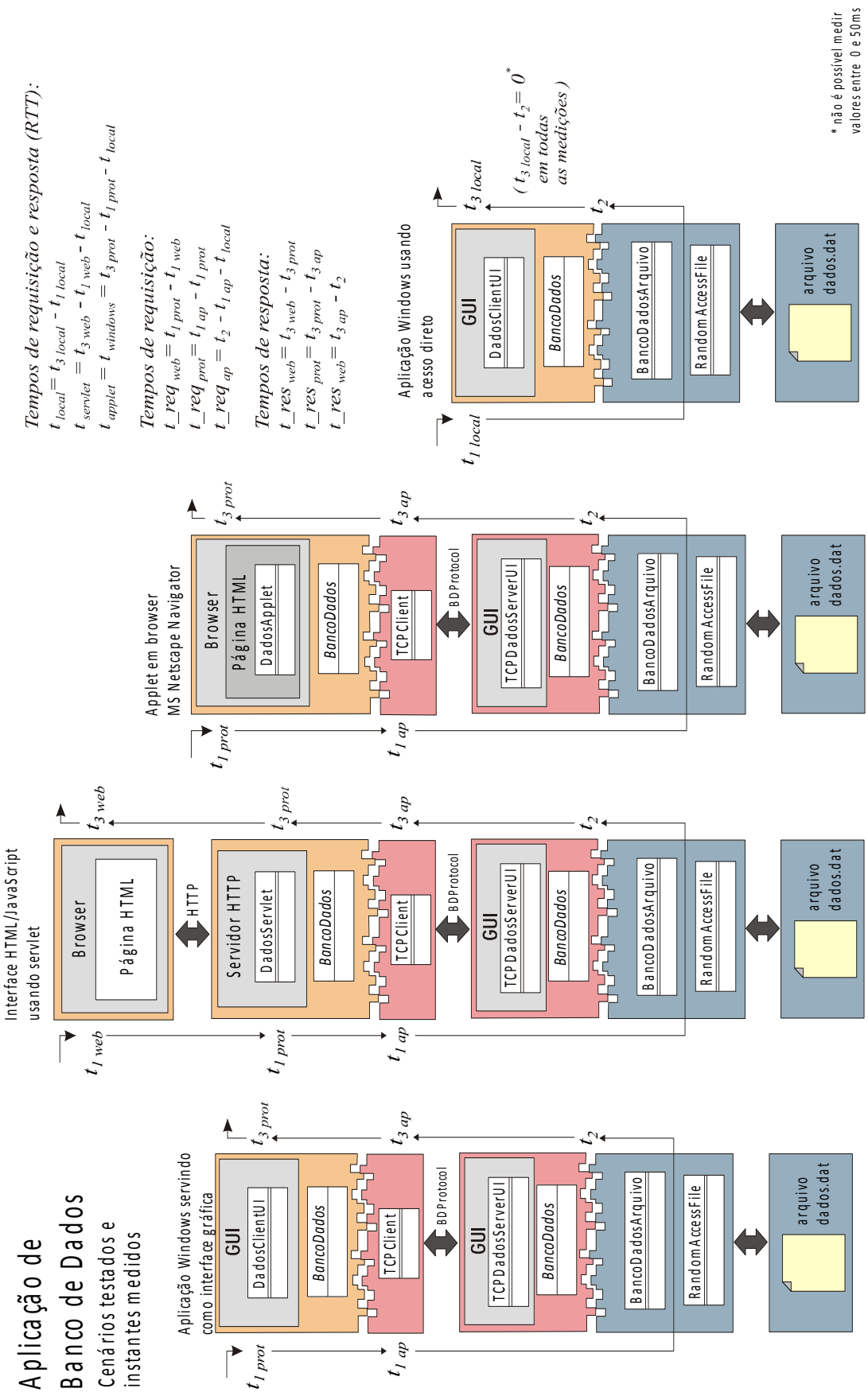


Figura 10-7 - Combinações dos blocos da aplicação "bancodados" utilizadas nos testes

10.3. Execução das aplicações

Os pacotes (módulos Java) desta aplicação são compartilhados tanto pelas aplicações cliente como pelas aplicações servidoras, portanto, para rodar a aplicação em uma máquina é preciso instalar todo o pacote, mesmo que apenas parte da aplicação seja usada. As aplicações analisadas neste módulo estão no diretório `/jad/apps/` criado após a descompactação do disquete distribuído juntamente com este texto.

As aplicações podem ser executadas invocando diretamente o interpretador Java ou através de roteiros `.BAT` (Windows) ou Bourne-Shell (Unix). Para executar qualquer aplicação é preciso iniciar uma classe Java executável (cujo nome termina em `UI`). Também é preciso definir uma propriedade que informa o diretório de trabalho da aplicação. As propriedades podem ser passadas via linha de comando através do argumento `-D` do interpretador Java. O `classpath` só precisa ser definido se a aplicação for executada fora do seu diretório de trabalho. A sintaxe geral para executar *qualquer* programa da aplicação de banco de dados é:

```
java -Dapp.home=/jad/apps -classpath c:/jad/apps bancodados.user.xxxUI
```

É mais fácil iniciar uma aplicação usando um dos roteiros de execução disponíveis, em MS-DOS (ou Bourne-Shell). Para usar os roteiros é preciso configurá-los para que contenham o endereço correto do interpretador Java e definam corretamente variáveis de ambiente e propriedades do sistema. As instruções de como configurar as aplicações estão presentes como comentários no código de cada roteiro e na documentação HTML da aplicação.

Os roteiros executáveis *Unix* têm extensão `.sh` e os executáveis *Windows* têm extensão `.lnk` ou `.bat`. As aplicações Web têm extensão `.html`. Todos os programas/páginas têm a forma `runXXX`. Pode não ser necessário configurá-los caso o diretório `/jad/` tenha sido instalado nas localidades *default* (`c:\` para *Windows* e `~/` para *Unix*). Os programas, localizados em `/jad/apps/`, são:

- `runRMIServer.bat` e `runRMIServer.sh` - Executa servidor RMI e `rmiregistry`
- `runRIIOPServer.bat` e `runRIIOPServer.sh` - Executa servidor RMI/IIOP e `tnameserv`
- `runTCPServer.bat` e `runTCPServer.sh` - Executa servidor BDProtocol TCP/IP

- `runCorbaServer.bat` e `runCorbaServer.sh` - Executa servidor CORBA e `tnameserv`
- `runConsole.bat` e `runConsole.sh` - Executa cliente orientado a caracter
- `runGraphic.lnk` e `runGraphic.sh` - Executa cliente gráfico Windows/X-Window
- `runApplet.bat` e `runApplet.sh` - Roda *Applet* no `appletviewer`
- `runWeb.html` - Página que inicia a aplicação *Applet* (pode necessitar de servidores) e a aplicação *Servlet* (necessita de `servletrunner` executando). URL *default* para servlet é: `http://localhost:8080/servlet/bdservlet`.
- `runServlet.bat` e `runServlet`.sh - Inicia servidor Web `servletrunner`

Toda a documentação sobre a aplicação, incluindo suas classes, interfaces, métodos, pacotes está no diretório `jad/apps/docs/` a partir do arquivo HTML `index.html`. O arquivo MS-DOS `genDocs.bat` gera a documentação caso esta não esteja disponível. Neste caso, é preciso criar um diretório `docs` abaixo de `jad/apps/`.

Maiores detalhes sobre a construção das aplicações estão nos módulos *Java 11*, *Java 6* e *Java 7*.

10.4.Construção da aplicação

Este apêndice fornece alguns detalhes sobre a estrutura da aplicação descrita na seção anterior. Maiores detalhes sobre o código em Java podem ser encontrados no diretório `/jad/apps/` (resultante da expansão dos arquivos do disquete) no código-fonte, detalhadamente comentado (`/jad/apps/bancodados/`) e na documentação em hipertexto, gerada a partir do código-fonte, que descreve todos os pacotes, classes e métodos (`/jad/apps/docs/`).

Este apêndice supõe que o leitor esteja familiarizado com a linguagem Java, HTML e JavaScript.

Estrutura da aplicação

A aplicação estudada nos módulos Java 6 e Java 7 permite o acesso a um banco de dados contendo registros de *anúncios*. Fornece diversas interfaces do usuário e opções de servidores intermediários. O núcleo da aplicação, porém, consiste apenas de dois *tipos* Java:

- **`bancodados.BancoDados`**: uma interface que define os métodos utilizados por qualquer objeto que implemente serviços de acesso ao banco de dados. Representa o banco de dados ou quadro de avisos (do sistema de anúncios).
- **`bancodados.Registro`**: classe que representa um registro do banco de dados (ou um anúncio).

Estes dois tipos estão armazenados em um pacote chamado `bancodados`. A figura 10-8 ilustra os diagramas da classe `Registro` e interface `BancoDados` e seus métodos.

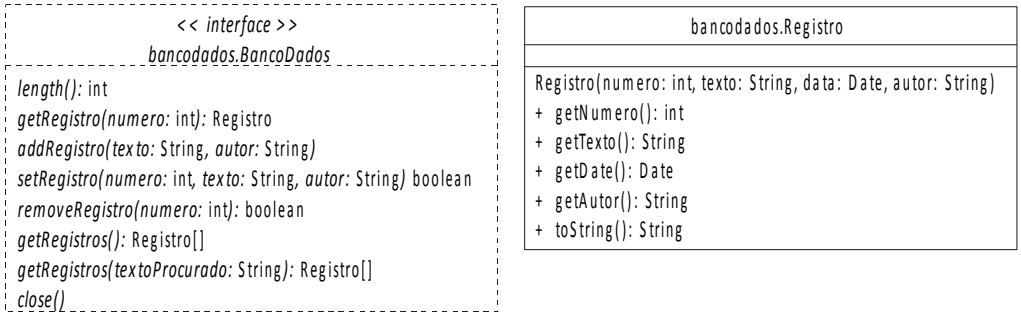


Figura 10-8 Diagramas de BancoDados e Registro

A interface `BancoDados` é utilizada por classes que implementam a interface do usuário. Através dela, todas as interfaces do usuário podem chamar métodos para realizar operações em um banco de dados sem precisar saber coisa alguma sobre sua estrutura interna ou sobre sua localização, pois a interface contém apenas a *assinatura* dos métodos.

Nesta aplicação, desenvolvemos oito diferentes interfaces do usuário (4 clientes e 4 aplicações intermediárias, operadas remotamente pelos clientes) que criam, removem, atualizam, pesquisam e recuperam registros de um banco de dados reutilizando a interface `BancoDados`. As classes que compõem as interfaces do usuário estão no pacote `bancodados.user`. Todas usam referências do tipo `BancoDados` através das quais realizam as operações solicitadas.

As classes que terminam em **`UI`** são executáveis (possuem método `main()` e podem ser executadas pelo sistema de tempo de execução Java). Classes que possuem o nome **`Client`** são usadas como componentes do cliente. Classes que possuem o nome **`Server`** são usadas como componentes das aplicações que implementam os servidores intermediários.

As aplicações que interagem com o usuário utilizam as classes:

- **`DadosClientTextUI`**: Interface do usuário orientada a caracter.
- **`DadosClientPanel`**: Interface gráfica do usuário. Esta não é uma classe executável. É um `java.awt.Panel` que é usado como parte de um objeto do tipo `DadosClientUI` ou `DadosApplet` oferecendo uma interface uniforme nos dois tipos de aplicação.
- **`DadosClientUI`**: `java.awt.Frame` que fornece a estrutura para que a interface gráfica do usuário (`DadosClientPanel`) possa ser usada como uma aplicação do *Windows*.
- **`DadosApplet`**: `Applet` que fornece uma estrutura para que a interface gráfica do usuário (`DadosClientPanel`) possa ser executada dentro de um browser.
- **`DadosServlet`**: `Servlet` que permite que o servidor Web atue como cliente para a aplicação e seja controlado por uma página HTML em um browser.

São quatro as aplicações usadas para implementar servidores intermediários. Elas agem como servidores e clientes ao mesmo tempo. Como servidores, recebem as requisições dos clientes. Como clientes, repassam as requisições à camada inferior, que pode ser outra aplicação intermediária ou um driver para o meio de armazenamento. A aparência gráfica de todas as aplicações é a mesma pois todas estendem uma classe que fornece essa estrutura:

- **`DadosServerFrame`**: Classe abstrata derivada de `Frame` que fornece uma interface gráfica de apresentação e métodos padrão para todos os servidores intermediários.
- **`RMIDadosServerUI`**, **`RMIIIOPDadosServerUI`**, **`CorbaDadosServerUI`**, **`TCPDadosServerUI`** Servidores intermediários que manipulam o banco de dados a partir de instruções remotas enviadas por clientes RMI/JRMP, RMI/IIOP (ou CORBA), CORBA e *BDProtocol* (protocolo proprietário), respectivamente.

Dependendo do tipo de servidor escolhido (nas aplicações cliente) as classes utilizadas poderão ser diferentes, como mostram as figuras 10-7 e 10-11, mas sempre preservam a estrutura de camadas mostrada na figura 10-9.

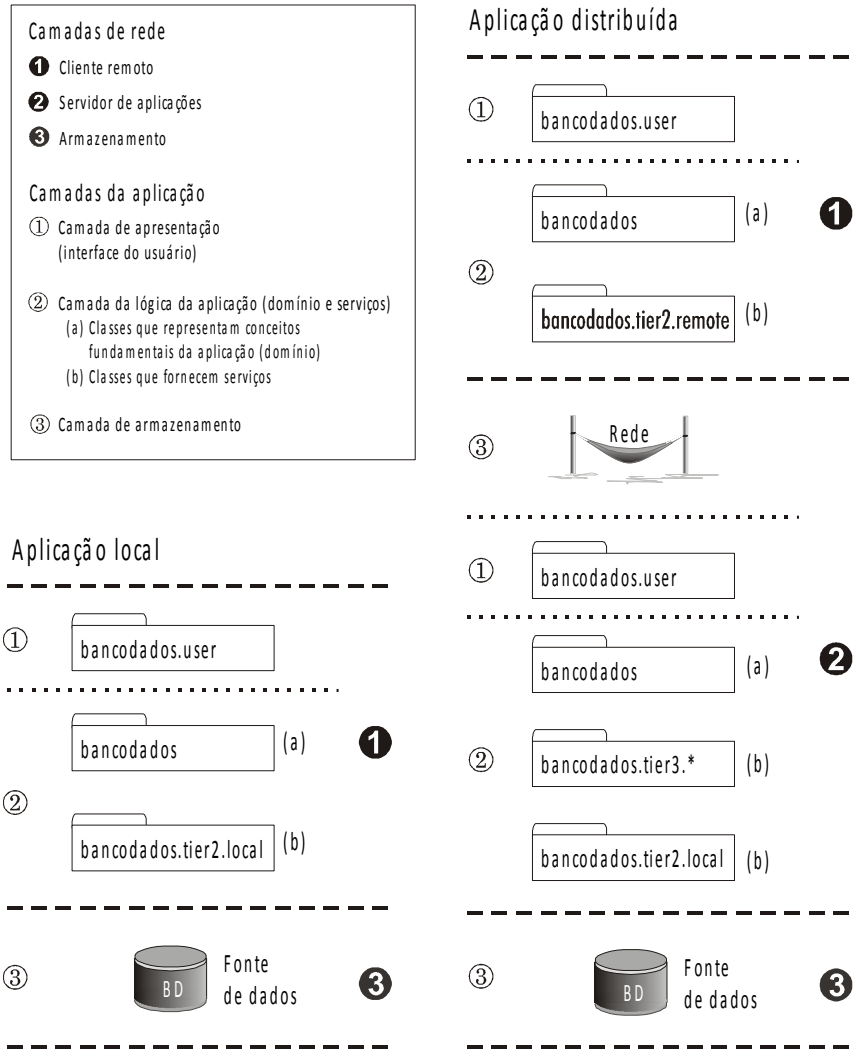
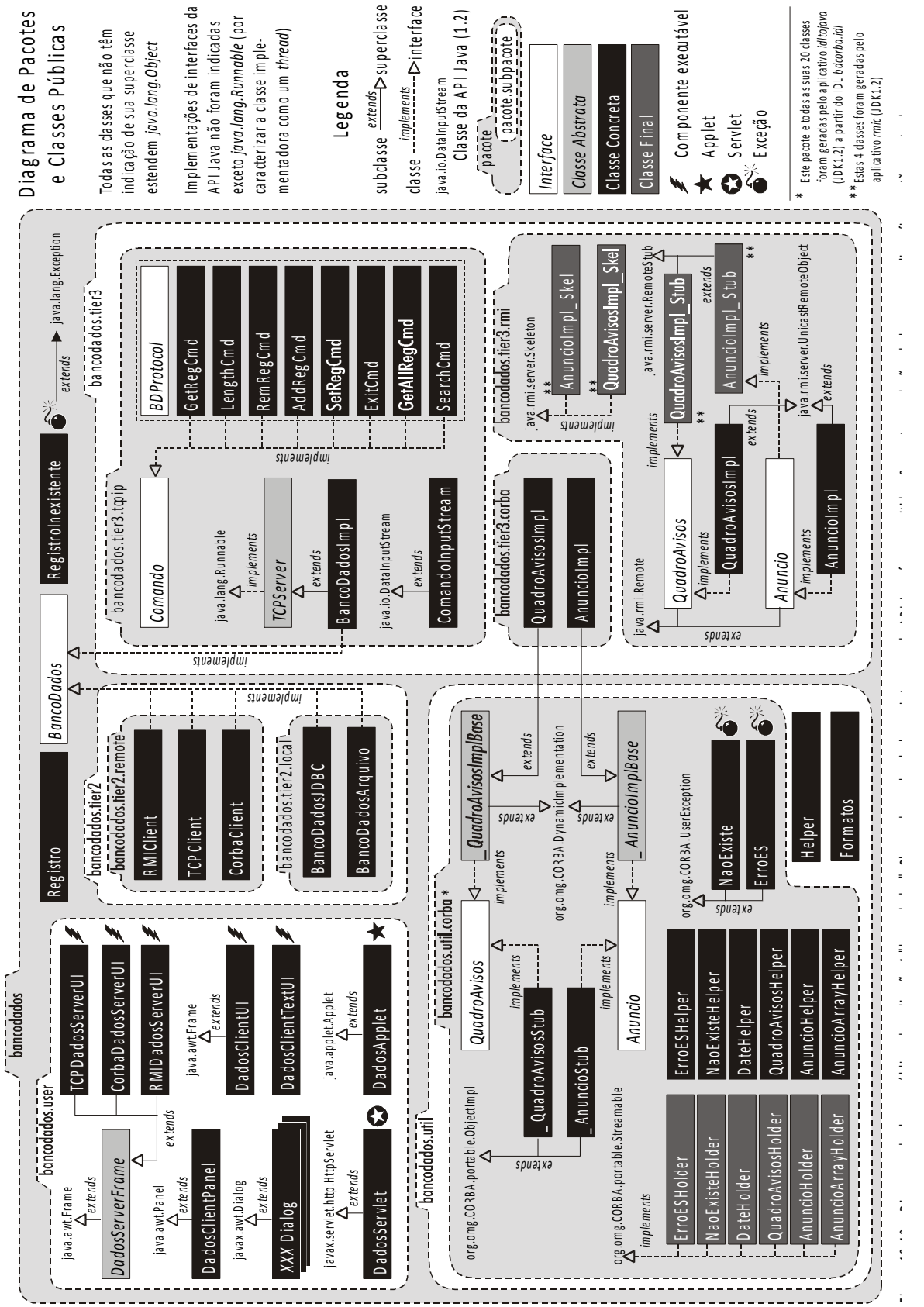


Figura 10-9 – Arquitetura em camadas e pacotes Java/UML das aplicações de banco de dados

As classes restantes do pacote `bancodados.user` são janelas de dialogo e adaptadoras de eventos usadas pelas aplicações gráficas. A figura 10-10 mostra todas as classes e todos os pacotes da aplicação.



A figura 10-11 mostra um cenário, de uma aplicação rodando como applet e usando CORBA para intermediar o acesso ao banco de dados.

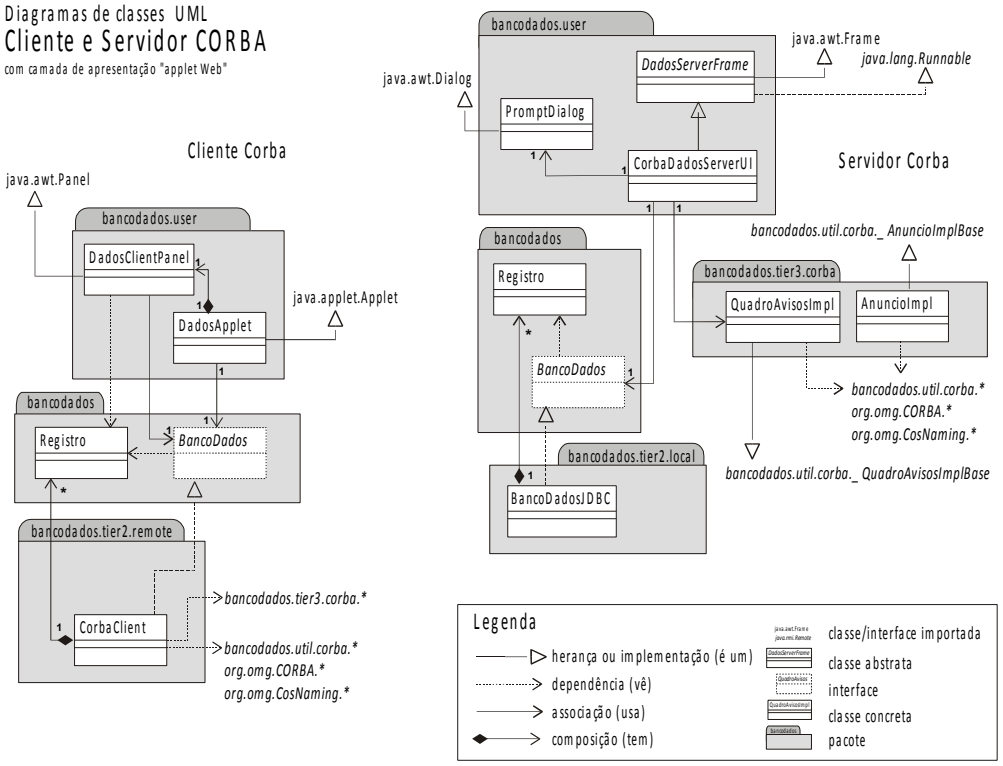


Figura 10-11 - Diagramas de classes. Cliente e Servidor CORBA com camada de apresentação "applet Web".

10.5.Estrutura do código: *camada de armazenamento*

Esta seção apresenta alguns detalhes sobre as classes que servem de "drivers" aos bancos de dados implementados em arquivo e em sistemas de bancos de dados relacionais.

Aplicação de banco de dados em arquivo

Para implementar o banco de dados precisamos implementar a interface `bancodados.BancoDados` (figura C-1). Cada método deve realizar suas operações sobre um objeto do tipo `java.io.RandomAccessFile` que armazenará os objetos do tipo `Registro` em disco. O registro tem o seguinte formato:

- **`int`**: número do anuncio
- **`String`**: texto do anuncio
- **`long`**: data (tempo em milissegundos desde 1/1/1970)
- **`String`**: autor do anuncio

Podemos usar os métodos da classe `RandomAccessFile`: `writeInt()`, `writeLong()` e `writeUTF()` para gravar os tipos `int`, `long` e `String`, respectivamente e `readInt()`, `readLong()` e `readUTF()` para recuperá-los posteriormente.

O banco de dados tem a seguinte organização no arquivo:

- Os registros serão acrescentados ao arquivo em seqüência.
- Cada novo registro será acrescentado no final do arquivo com um número igual ao maior número pertencente a um registro existente mais um, ou `100`, se não houver registros;
- Registros removidos terão o seu número alterado para `-1` (continuarão ocupando espaço no arquivo).
- Registros alterados serão primeiro removidos e depois acrescentados no final do arquivo com o mesmo número que tinham antes (também continuarão ocupando espaço no arquivo).

A classe que desenvolvemos está em `/jad/apps/bancodados/tier2/local/` e chama-se `BancoDadosArquivo.java`. Implementa `BancoDados` podendo ser utilizada por qualquer outra classe que manipule com essa interface. A classe possui um objeto `RandomAccessFile` que representa o arquivo onde os dados serão armazenados. Suas variáveis membro e a implementação de seu construtor estão mostrados abaixo:

```
public class BancoDadosArquivo implements BancoDados {

   private RandomAccessFile arquivo;   // descritor de arquivo
   private boolean arquivoAberto;      // inicialmente false
   private Hashtable bancoDados;       // relaciona posicao do ponteiro
                                       // do RandomAccessFile com registro
   private int maiorNumReg = 0;        // Maior número de registro

   public BancoDadosArquivo(String arquivoDados) throws IOException {
      try {
         arquivo = new RandomAccessFile(arquivoDados, "rw");
         arquivoAberto = true;
      } catch (IOException e) {
         close();
         throw e;    // propaga excecao para metodo invocador
      }
   }
(...)
```


A referência **`arquivo`** é utilizada em todos os métodos que manipulam com os dados no arquivo. Abaixo listamos o método `addRegistro()`, que adiciona um novo registro.

```
public synchronized void addRegistro(String anuncio, String contato) {
      try {
         arquivo.seek(arquivo.length());  // posiciona ponteiro no fim
         arquivo.writeInt(getProximoNumeroLivre());
         arquivo.writeUTF(anuncio);
         arquivo.writeLong(new Date().getTime());
         arquivo.writeUTF(contato);
      } catch (IOException ioe) {
         ioe.printStackTrace();
      }
   }
```

Para remover um registro, é preciso saber em que posição ele está. O `Hashtable bancoDados` (definido em `getRegistros()`) contém um mapa que relaciona o número do registro com a posição no arquivo. O método `removeRegistro()` utiliza então esta informação para localizar o registro que ele deve marcar como removido.

```
public synchronized boolean removeRegistro(int numero)
                                       throws RegistroInexistente {
      try {
        getRegistros();
        String pointer = (String)bancoDados.get(new Integer(numero));
        if (pointer == null)
           throw new RegistroInexistente("Registro não encontrado");
        long posicao = Long.parseLong(pointer);
        arquivo.seek(posicao);
        int numReg = arquivo.readInt();
        if (numReg != numero)
           throw new RegistroInexistente("Registro não localizado");
        arquivo.seek(posicao);
        arquivo.writeInt(-1);   // marca o registro como removido
        arquivo.seek(0);
      } catch (IOException ioe) { // (...)
      }
      return true;
   }
```

Nesta interface que desenvolvemos para o arquivo usando `RandomAccessFile`, os registros removidos nunca são realmente removidos. Para limpar o arquivo, livrando-o de espaço ocupado inutilmente, é preciso exportar todos os registros válidos e importá-los de volta em um novo arquivo.

Consulte o código fonte para detalhes sobre os outros métodos.

Construção de uma aplicação JDBC

Nesta seção, construímos uma aplicação JDBC usando os mesmos dados do capítulo anterior, desta vez organizado em um BD relacional. Para reutilizar toda a interface do usuário e as classes que representam os conceitos fundamentais do programa, criamos uma classe que implementa a interface `bancodados.BancoDados`. Como a interface do usuário usa a interface `BancoDados`, podemos usar a classe `bancodados.tier2.local.BancoDadosJDBC`, preservando a mesma interface do usuário usada para a versão baseada em arquivo.

Os dados utilizados por esta aplicação são do mesmo tipo que aqueles manipulados pela aplicação da seção anterior. Teremos, portanto, apenas uma tabela no banco de dados com a seguinte estrutura:

Tabela 10-1 – Estrutura do banco de dados

Coluna	Tipo de dados das linhas	Informações armazenadas	Observações
`codigo`	`int`	número do anúncio	`integer` *chave primária*
`data`	`String`	data de postagem do anúncio	`char(24)`
`texto`	`String`	texto do anúncio	`char(8192)`
`autor`	`String`	autor do anúncio	`char(50)`

Utilizamos os tipos de dados mais fundamentais para garantir a compatibilidade com uma quantidade maior de bancos de dados.

Na classe `BancoDadosJDBC` carregamos um driver de acordo com a URL passada pelo usuário, que permitirá, no final, obter um objeto `java.sql.Statement` (JDBC) através do qual poderemos enviar requisições SQL ao servidor. O método `executeUpdate`, da interface `Statement`, pode ser usado para inserir registros na implementação de `addRegistro()`:

```
public synchronized void addRegistro(String texto, String autor) {
      (...)
      String insert = "INSERT INTO anuncios VALUES (" + numero + ", '"
                                                      + quando + "', '"
                                                      + texto +  "', '"
                                                      + autor +  "')";
      try {
         stmt.executeUpdate(insert); // objeto tipo Statement
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
```


O método `getRegistros()` obtém uma tabela de dados como resposta a uma requisição `SELECT` (SQL) enviada pelo método `executeQuery()`. Esse método retorna um `ResultSet` (que contém uma tabela virtual navegável *linha-a-linha* via seu método `next()`). Para cada posição, usamos os métodos `get`*Tipo*`()` apropriados para ler inteiros e strings.

```
public Registro[] getRegistros() {
     ResultSet rs;
     Vector regsVec = new Vector();
     String query = "SELECT numero, data, texto, autor " +
                    "FROM anuncios ";
     try {
        rs = stmt.executeQuery(query);
        while (rs.next()) {
           int numero = rs.getInt("numero");
           String texto = rs.getString("texto");
           String dataStr = rs.getString("data");
           java.util.Date data = df.parse(dataStr);
           // java.util.Date data = rs.getDate("data");
           String autor = rs.getString("autor");
           regsVec.addElement(new Registro(numero, texto, data, autor));
        }
     } catch (SQLException e) { // (...)
     } catch (java.text.ParseException e) { // (...)
     }
     Registro[] regs = new Registro[regsVec.size()];
     regsVec.copyInto(regs);
     return regs;
  }
```

A remoção do registro é implementada de forma mais simples ainda, bastando encapsular uma instrução SQL `DELETE`:

```
public synchronized boolean removeRegistro(int numero)
      throws RegistroInexistente {

      ResultSet rs;
      String delete = "DELETE FROM anuncios WHERE numero = " + numero;

      try {
         stmt.executeUpdate(delete);
         return true;
      } catch (SQLException e) {
         e.printStackTrace();
         return false;
      }
  }
```


10.6.Estrutura do código: *interfaces do usuário*

Nesta seção apresentamos detalhes referentes ao funcionamento da interface do usuário orientada a caractere (que contém as mesmas operações básicas presentes nas interfaces do usuário baseadas em applet, servlet e aplicação *Windows*) e das interfaces HTML, geradas pelo servlet.

Interface orientada a caractere

Os arquivos utilizados nesta aplicação estão nos subdiretórios a seguir. Em negrito está o único arquivo que trata desta interface:

Tabela 10-2 – Componentes da aplicação de banco de dados, com acesso local apenas.

Subdiretório	Arquivo-fonte Java	Conteúdo
`bancodados/user`	**`DadosClientTextUI.java`**	interface do usuário orientada a caracter
`bancodados/`	`BancoDados.java`	interface genérica para o banco de dados (interface)
`bancodados/`	`Registro.java`	representação de um registro (classe concreta)
`bancodados/tier2/local`	`BancoDadosArquivo.java`	implementação de `BancoDados`
`bancodados/tier2/local`	`BancoDadosJDBC.java`	implementação de `BancoDados`

A interface do usuário deve manipular com um objeto `BancoDados`. Na prática, estará manipulando com o `RandomAccessFile` através da classe `BancoDadosArquivo` ou com os métodos JDBC através da classe `BancoDadosJDBC`, mas ela não precisa saber disso. Se estiver usando uma aplicação intermediária, poderá estar usando um cliente CORBA ou RMI que implementa `BancoDados`. Resumindo, a interface do usuário é uma camada que independe da forma de organização ou da localização dos dados que manipula.

A classe `DadosClientTextUI` declara uma variável membro do tipo `BancoDados`:

```
private BancoDados client;
```

e em todos os seus métodos chama métodos de `BancoDados` através de `client`. Apenas o menu principal refere-se ao `BancoDadosArquivo` ou `BancoDadosJDBC`.

Quando o usuário escolhe um dos dois, ele é instanciado e sua referência é passada para `client`. A partir daí, todos os métodos operam sobre a interface `BancoDados`.

Se o usuário decidir criar um novo registro, por exemplo, a aplicação chamará o método local `criar()`, que contém:

```
public void criar() throws IOException {
      (...)
      client.addRegistro(texto, autor); // método de BancoDados
   }
```

Para listar todos os registros, o método `mostrarTodos()` é chamado:

```
public void mostrarTodos() throws IOException, RegistroInexistente {

      (...)
      Registro[] regs = client.getRegistros();
      (...)
}
```

Em *nenhum* dos métodos há indicações que acontece alguma coisa em um `RandomAccessFile` (banco de dados baseado em arquivo) ou na interface `Statement` (banco de dados relacional), portanto, a camada de apresentação está isolada da segunda camada.

Interface HTML com servlets HTTP

Com servlets, a aplicação de banco de dados pode ser acessível através de uma interface HTML. Construímos uma interface baseada em duas páginas. A primeira contém a interface onde o usuário pode escolher endereço e serviço que irá fornecer os dados. Passando da primeira página (um serviço foi selecionado e este aceitou a conexão), uma segunda página será mostrada com todos os registros disponíveis no banco de dados[1]. A primeira página não é alterada pelo servlet. É simplesmente lida do disco e repassada ao browser.

Um esqueleto da segunda página deve ser lido pelo servlet que irá preencher uma tabela com todos os registros encontrados antes de enviá-la para o browser. Este preenchimento também inclui uma lista de vínculos (links) de acesso rápido, antes e depois da tabela, e vínculos rápidos para o menu em qualquer registro. No início da segunda página há um painel de controle que permite gerenciar o

[1] Não foram tomadas providências para quebrar a página em páginas menores a medida em que o número de registro crescer, portanto, esta versão pode ser impraticável para acessar grandes quantidades de informação.

banco de dados. Esses botões respondem a eventos JavaScript e alteram as informações que serão enviadas para o servlet. No final da página há uma área para edição e criação de novos registros, construídos usando formulários HTML. As páginas estão localizadas no diretório `/jad/apps/htdocs/`. A página foi criada para ser carregada via servlet (a carga direta do arquivo via browser provocará erros de JavaScript).

O código da página dados.html antes de ser lida pelo servlet contém marcadores de posição indicados por comentários HTML (**`<!-- -->`**):

```
(...)
<!-- links -->
</p><table border=0 width=590 bgcolor="#dddddd">
<tr><th>Número</th><th>Texto</th><th>Autor</th><th>Data</th></tr>
<!-- dados -->
(...)
<tr><td width=70> </td><td> </td><td> </td><td> 
</td></tr></table><p>
<!-- links -->
</p></div>
(...)
```

Após a leitura da página pelo servlet, o browser recebe uma página gerada dinamicamente, com os marcadores `<!--links -->` e `<!--dados -->` (em negrito no código acima e abaixo) substituídos por informações geradas a partir dos dados armazenados (veja figura 10-4):

```
<a href="#100">100</a> | <a href="#101">101</a>
 (... várias linhas removidas ...)
| <a href="#134">134</a> |
</p><table border=0 width=590 bgcolor="#dddddd">
<tr><th>Número</th><th>Texto</th><th>Autor</th><th>Data</th></tr>
<tr valign=top><td><a name="100">100</br><a
href="#top"><small>[MENU]</small></a></td><td>Cachimbos, ...
também!</td><td>R. Magritte
(magritte@recursive.org)</td><td>21/12/97 18:39</td></tr>
 (... várias linhas removidas ...)
<tr valign=top><td><a name="131">131</br><a
href="#top"><small>[MENU]</small></a></td><td>Vende-se um ... da
Terra.</td><td>G. Galileo (galileo@galilei.lua)</td><td>31/12/98
17:06</td></tr>
<input type=hidden name=tarea value="">
<input type=hidden name=tfld value="">
<input type=hidden name=numero value="0">
(...)
<tr><td width=70> </td><td> </td><td> </td><td> 
</td></tr></table><p>
<a href="#100">100</a> | <a href="#101">101</a>
 (... várias linhas removidas ...)
| <a href="#134">134</a> |
</p></div>
```


Para implementar a interface do usuário, podemos usar somente HTML. A vantagem é que nossa página será acessível até pelo mais primitivo dos browsers. O problema é que HTML é muito limitado quanto aos recursos de interação com o usuário. HTML oferece três tipos de eventos:

- "clicar sobre um link" que inicia uma requisição `GET` ao servidor.
- "apertar um botão submit" que envia os dados de um formulário ao servidor através de uma requisição `POST` ou `GET`
- "apertar um botão reset", que reinicializa os campos de um formulário aos seus valores *default*.

Os controles da aplicação de banco de dados são mais complexos. É preciso que os botões façam mais que simplesmente enviar dados ao servidor. Pode ser que o usuário da aplicação Web selecione um registro e queira apagá-lo ao apertar o botão. Pode ser que ele selecione um registro e aperte o botão para alterá-lo. Pode ser que ele queira realizar uma busca.

Para superar essa e outras limitações do HTML, usamos JavaScript. JavaScript possui cerca de 13 eventos e botões podem ser reprogramados para realizarem algo diferente de limpar campos ou enviar dados. Para programá-los, usamos botões neutros (que não provocam eventos). Estes botões são representados em HTML por:

```
<input type="button">
```

O evento de clique do botão é representado pelo atributo `onclick` que pode ser usado em qualquer botão e contém instruções JavaScript que devem ser executadas assim que o botão for apertado.

```
<input type=button onclick="alert('O botão foi apertado!')">
```

Na nossa interface, fizemos com que alguns botões chamassem funções, definidas no bloco `<script> ... </script>` no início do arquivo, e outros mudassem *parâmetros ocultos*, representados em HTML como

```
<input type=hidden name="variavel" value="valor da variavel">
```

Os dados dos campos ocultos são passados na requisição do browser da mesma maneira em que são passados os dados de campos de texto e outros dispositivos de entrada. O poder do JavaScript está na possibilidade de mudar o

valor dos campos ocultos enquanto uma página está sendo exibida, em resposta a um evento (um aperto de botão, por exemplo).

Usamos campos ocultos, por exemplo, para que o servlet saiba qual o 'comando' que foi solicitado pelo usuário da aplicação Web, alterando o campo

```
<input type="hidden" name="comando" value="getReg">
```

Quando o usuário aperta o botão "Remover" para remover um registro, o conteúdo do atributo `onclick` desse botão é executado:

```
<INPUT TYPE="button" NAME="tira" value="Remover..."
                     onclick="remover(this.form)">
```

`remover()` é o nome de uma função JavaScript definida no início da página. `this.form` é uma referência que passa o próprio formulário como argumento da função. A função `remover()` está definida como:

```
<script>
function remover(entra) {
  candidato = prompt("Digite número do registro a ser removido","");
  if (candidato) {
    entra.numero.value = candidato;  // muda valor de campo oculto 'numero'
    if (confirm("Tem certeza que quer remover o registro " + candidato + "?")) {
       entra.comando.value = "remReg"; // muda valor do campo 'comando'
       entra.submit();
    }
  }
}
</script>
```

Dentro da função, o formulário é representado pela variável **`entra`**. `entra.comando` é uma referência ao campo oculto de nome `comando`. `entra.comando.value` é o valor deste campo que na linha marcada em negrito acima é alterado de '`getReg`' para '`remReg`'. A linha seguinte

```
entra.submit();
```

envia o formulário. O servlet, após decodificar a linha de dados recebida, buscará pelo nome '`comando`' e receberá o valor '`remReg`', indicando que o usuário deseja remover um registro. O número do registro a ser removido foi armazenado em outro campo oculto que o servlet pode ler. Desta forma, é possível implementar todas as outras funções da interface do usuário.

Para maiores detalhes sobre esta aplicação, consulte o código fonte localizado no arquivo `DadosServlet.java` (diretório `jad/apps/bancodados/user`).

Os exemplos deste capítulo foram testados usando o *Servletrunner*. Para executar os servlets, estes precisam ser instalados no *Servletrunner*. A instalação é realizada através de um arquivo de configuração chamado `servlet.properties` que vincula o nome do servlet a um arquivo `.class` e passa quaisquer parâmetros adicionais de inicialização necessários. Para esta aplicação, o arquivo `servlet.properties` deverá conter os seguintes dados:

```
servlet.bdservlet.code=bancodados.user.DadosServlet
servlet.bdservlet.initArgs=\
        htmlDados=j:/jad/apps/htdocs/dados.html,\
        htmlSelecao=j:/jad/apps/htdocs/selecao.html,\
        appHome=j:/jad/apps
```

O *Servletrunner* pode ser iniciado para esta aplicação rodando o arquivo `runServlet.sh` (ou `runServlet.bat`) no subdiretório `/jad/apps/`. Depois de iniciado, o *Servletrunner* estará no ar na porta 8080 (*default*) e irá servir servlets armazenados em (ou localizáveis a partir de) `jad/apps/`. A URL para chamar o servlet `bdservlet` é `http://`*`servidor`*`:8080/servlet/ bdservlet`. O nome do servidor e a sua porta devem ser os nomes e porta de onde o *Servletrunner* está rodando.

10.7.Estrutura do código: *aplicações intermediárias*

Não há grandes diferenças entre a parte cliente dessas aplicações e as camadas de armazenamento, do ponto de vista da camada de apresentação. Todas implementam a interface `bancodados.BancoDados`. Seu código fonte se está em `/jad/apps/bancodados/tier2/ remote/`.

Cada servidor, porém, tem uma estrutura própria de acordo com a tecnologia que utiliza. Precisam rodar como processos ativos para que possam ficar aguardando clientes. Todos os servidores possuem uma interface gráfica em `bancodados.user` que estende a classe abstrata `DadosServerFrame`. Ela possui a infraestrutura básica para qualquer servidor e permite que o cliente escolha um arquivo ou uma fonte de dados JDBC que o servidor irá servir.

O pacote `bancodados.tier3` contém um sub-pacote para cada implementação de servidor. Para maiores informações sobre essas aplicações, consulte o código-fonte em `/jad/apps/bancodados/tier3/`.

10.8. Onde e quando usar cada cenário?

Applets e Servlets

Usar um applet como camada de apresentação para uma aplicação localizada em um servidor remoto oferece as seguintes vantagens em relação a tecnologias baseadas no servidor:

- *Interface gráfica com mais funções e recursos*. Os recursos gráficos e interativos do HTML são limitados. Não é possível, por exemplo, redesenhar uma área da tela, trocar um formulário por outro ou fazer aparecer um texto na tela sem carregar uma nova página. Uma aplicação de pintura, que roda como um applet, é exemplo de uma aplicação que não poderia ser implementada apenas com HTML, JavaScript e servlets. Também conseguimos manter praticamente a mesma interface usada na aplicação de banco de dados, versão *Windows*, na versão applet, o que não seria praticável com servlets/HTML.
- *Resposta mais rápida a ações locais*. Usando applets, tarefas como navegação no banco de dados, mudança entre os modos de edição e navegação entre os registros do banco são mais rápidas e eficientes. Na versão HTML/servlets precisamos fazer uma chamada ao servidor para mudar de modo e qualquer alteração implica na carga de uma nova página HTML. Mesmo que o tempo seja menor, o usuário sempre tem a impressão que o tempo é maior, já que a interface da aplicação some por uns instantes (enquanto a página é carregada).
- *Flexibilidade em relação ao protocolo de transferência de dados*. A solução HTML/servlet está presa ao protocolo HTTP que intermedeia toda a sua comunicação. Usando um applet, o protocolo HTTP será usado apenas para transferir a aplicação para o browser. Depois que o applet estiver executando, poderá usar um outro protocolo aberto ou proprietário para se comunicar com o servidor [SUN95].

- *Flexibilidade em relação aos tipos de dados suportados*. Os browsers que utilizamos só suportam a exibição de imagens GIF, JPEG e PNG. Applets podem ser construídos para que suportem outros formatos proprietários [SUN 95].

Applets, porém, têm limitações. O uso de servlets para intermediar o acesso a uma aplicação remota e uma interface do usuário baseada em HTML e JavaScript permite realizar algumas tarefas difíceis ou impossíveis para os applets:

- *Interface do usuário disponível imediatamente*. A interface do usuário de uma aplicação baseada no servidor é uma página HTML, que geralmente é carregada rapidamente. Applets geralmente são maiores e levam tempo para iniciarem e aparecerem na tela.
- *Menos problemas de compatibilidade*. Aplicações baseadas no servidor podem ser utilizadas por um público-alvo mais amplo. Suporte total a Java é raro até nos browsers mais recentes. Uma aplicação baseada no servidor está praticamente imune a incompatibilidades entre versões e fabricantes de browser. O applet de banco de dados só roda sem problemas em versões mais recentes dos browsers comerciais.
- *Menos restrições de segurança*. Existem várias restrições de segurança associadas aos applets. Applets não podem[2], por exemplo, ter acesso máquinas da rede que não sejam a máquina de onde vieram. Servlets, como rodam no servidor e não no browser, estão livres desta restrição.

Applets e servlets não são tecnologias concorrentes. Podem ser usadas em conjunto com grandes benefícios, aproveitando as vantagens de ambos.

Clientes Web e clientes nativos

O objetivo desta discussão é apontar as principais diferenças entre aplicações Web, executando em browsers, e aplicações *nativas,* executando em sistema operacional nativo (Windows, por exemplo). As duas formas são aplicadas em situações diferentes. Podemos utilizar os resultados quanto ao desempenho para descobrir quando vale a pena fazer o *download* da aplicação para instalação local (e acesso remoto) em vez de usar a interface proporcionada pelo applet dentro do browser.

[2] É possível reduzir as restrições de segurança usando applets assinados.

A possibilidade de rodar uma aplicação dentro de um browser é um dos principais avanços proporcionados pela linguagem Java ao ambiente Web. As vantagens são muitas:

- *Facilidade de distribuição.* Na forma de um applet, a interface cliente da aplicação poderá ser distribuída facilmente através de uma Intranet ou da Internet, bastando que o usuário acesse a URL da página onde o applet está localizado.
- *Facilidade de atualização.* A qualquer momento a aplicação pode ser atualizada. Na próxima vez que um usuário solicitar o applet, ele já terá a última versão.
- *Facilidade de uso e instalação.* Não é preciso instalar o applet. Tendo-se um browser, é só carregá-lo.
- *Disponibilidade imediata.* O applet está imediatamente disponível. Não é preciso obter drivers externos ou instalar ambientes de execução. Os principais browsers do mercado oferecem um ambiente de execução nativo.
- *Segurança embutida.* Applets descarregados pela rede são sempre verificados pelo browser e não têm acesso ao sistema de arquivos local. Para eliminar essas restrições e ainda assim operar em um ambiente seguro, pode-se assinar applets digitalmente e fazer uso dos recursos de criptografia e autenticação disponíveis em Java.

Apesar de todas as vantagens, o ambiente Web ainda não segue um padrão bem definido. Os browsers apresentam incompatibilidades, são pouco eficientes e podem impor restrições em excesso. Rodar uma aplicação sobre o sistema operacional nativo, portanto, pode ser uma opção já que existem ambientes de execução Java para os principais sistemas operacionais, permitindo que o programa rode em *Windows*, *Unix*, *Macintosh*, etc. mesmo fora de um browser. Um usuário pode instalar a plataforma Java em sua máquina e rodar a aplicação cliente localmente, quando quiser. O browser seria usado apenas uma vez, para fazer o download da aplicação. As principais vantagens desse modelo sobre os applet são:

- *Controle sobre restrições de segurança.* As restrições impostas aos applets pelos gerentes de segurança dos browsers podem ser excessivas. Não é

possível, por exemplo, fazer com que um applet[3] imprima, salve um arquivo temporário em disco local, ou realize conexões a outras máquinas da rede [GOSL96]. Com uma aplicação independente, podemos implementar um gerente de segurança mais flexível, com menos restrições.

- *Maior velocidade*. Verificamos que a aplicação de banco de dados rodando como uma aplicação *Windows* apresentou um tempo de resposta e requisição bem menor àquele obtido com a mesma aplicação, local, usando um applet. Isto não leva em conta o tráfego na rede, já que a medição foi obtida com o acesso local. O gerente de segurança e o próprio browser contribuem para o baixo desempenho do applet.
- *Independência de fabricante de browser*. Poucos browsers suportam CORBA, Swing, Java2D, RMI sobre IIOP e outros recursos que só recentemente passaram a fazer parte da plataforma Java. Se browser algum suporta um recurso essencial de uma aplicação, será necessário que ele descarregue toda a API que contém as classes usadas pelo recurso, todas as vezes em que for executado.
- *Possibilidade de oferecer mais recursos*. Applets são construídos sob um regime de austeridade. Precisam ter o menor tamanho possível e freqüentemente evitar usar novas e eficientes APIs, por falta de suporte dos browsers. Com uma aplicação nativa, é possível incluir recursos melhores, utilizar APIs proprietárias mesmo que isto resulte em uma aplicação maior. O tamanho é menos crítico pois o produto só será descarregado uma vez, e depois será instalado localmente.

10.9.Resumo

Este módulo analisou um exemplo de aplicação Java que utiliza as tecnologias exploradas em módulos anteriores. Também comparou diferentes implementações da camada de apresentação (interface do usuário) de uma mesma aplicação, que pode rodar como applet, em um browser Web; como aplicação independente, sob o sistema operacional *Windows*; ou como servlet, em um servidor Web gerando páginas dinamicamente para um browser.

[3] É possível reduzir as restrições de segurança usando applets assinados.

Os resultados da análise fornecem subsídios que podem orientar decisões para usar uma ou outra interface do cliente. As discussões e comparações realizadas neste capítulo estão resumidas na tabela abaixo, para consulta rápida.

Tabela 10-3

	Cliente Windows	**Cliente applet**	**Cliente HTML + servlet HTTP**
Interface gráfica	Completa e interativa *(usa todos os recursos do pacote java.awt)*	Completa e interativa *(usa todos os recursos do pacote java.awt)*	Limitada *(interatividade depende de programação adicional em JavaScript)*
Tempo de resposta da GUI (resposta a eventos locais)	Melhor tempo de resposta *(GUI nativa)*	Melhor tempo de resposta *(GUI nativa)*	Pior tempo de resposta *(precisa carregar nova página)*
Tempo de resposta de operações de rede	Melhor tempo de resposta	Tempo adicional devido ao browser	Tempo adicional devido ao protocolo HTTP
Protocolos suportados	Qualquer um	Qualquer um	HTTP
Formatos que podem ser exibidos (tipos de dados)	Qualquer tipo	Qualquer tipo	HTML, GIF, JPEG, PNG e tipos suportados pelo browser usado
Recursos e APIs Java suportadas pela aplicação	Todos os recursos disponíveis	Apenas recursos disponíveis no browser	Todos os recursos disponíveis
Segurança	Restrições definidas pelo programador	Restrições impostas pelo browser (ñ-assinados)	Restrições definidas pelo programador e servidor
Facilidades de distribuição, atualização e instalação	Usuário precisa instalar (uma vez) e atualizar.	Instalação automática após carga pelo browser, a cada acesso.	Instalação prévia no servidor Web. Usuário simplesmente usa o serviço.
Flexibilidade de desenvolvimento[4]	Total (por ser aplicação de referência).	Limitada por herança (precisa estender a classe Applet).	Limitada por polimorfismo (precisa implementar interface Servlet)[5].
Facilidade de desenvolvimento[6]	Maior (referência).	Mesmo nível que aplicação de referência.	Requer conhecimentos de HTML e HTTP.
Quantidade de modificações e código adicional a escrever[7]	Nenhum (referência).	Poucas modificações (usa mesma interface gráfica).	Algumas modificações (precisa gerar HTML).
Suporte em JDK 1.2 (Java 2)	java.awt e javax.swing (ambos do núcleo JDK)	java.applet (núcleo)	javax.servlet (extensão)
Suporte em JDK 1.1	java.awt (núcleo) e com.sun.java.swing (extensão)	java.applet (núcleo)	javax.servlet (extensão)
Suporte em JDK 1.0	java.awt (núcleo)	java.applet (núcleo)	Não suportado.

[4] Necessidade de seguir regras rígidas, como herança de certas classes, implementação de certos métodos, utilização dentro de certos parâmetros. Herança (extensão de classes) é bem menos flexível que polimorfismo sem herança (implementação de interfaces) uma vez que Java não suporta herança múltipla de implementações, mas permite que uma classe implemente várias interfaces.

[5] Tipicamente implementam-se servlets HTTP através de herança, estendendo a classe HttpServlet que por sua vez implementa a interface Servlet (polimorfismo sem herança).

[6] Leva em consideração número de linhas de código adicionais (em relação à aplicação *Windows*) que precisam ser escritas, linguagens e tecnologias que devem ser conhecidas (como HTML, HTTP). Não leva em conta a necessidade de se aprender a API específica para cada tecnologia.

[7] Em relação à aplicação *Windows*.

Na plataforma Web, tanto applets, servlets ou ambos podem ser usados. Servlets são indicados quando a aplicação requer comunicação com outras aplicações ou informações localizadas na máquina servidora, e quando uma aplicação necessita atingir um público-alvo amplo, que não pode ser excluído por não possuir um browser de última geração.

Applets são recomendados nas aplicações Web em que os clientes possuem browsers de última geração e quando a aplicação requer grande interação em tempo real no cliente (desenhos, por exemplo). O inconveniente é o download de um programa grande e a falta de suporte por alguns browsers. As restrições podem ser atenuadas por applets assinados.

O uso de aplicações independentes (sem usar browsers ou servidores HTTP) permite que se ofereçam mais serviços, interatividade e velocidade de acesso maior que applets e servlets, mas traz o inconveniente de requerer *download* e *instalação* por parte do cliente (o que é raro em aplicações Web, mesmo em intranets).

Fonte:

Helder L. S. da Rocha. *Dissertação de Mestrado. Apêndice C.* Universidade Federal da Paraíba, Campus de Campina Grande, 1999.

