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.

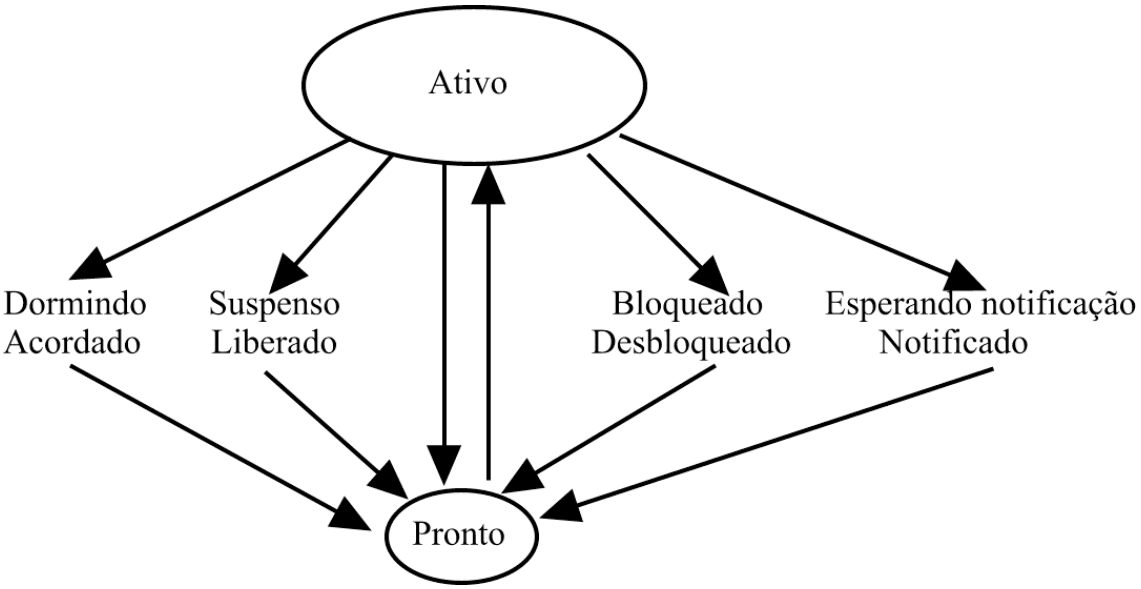
O método `setPriority()` pode ser chamado para definir a prioridade de threads. Os valores passados podem ser qualquer valor de 1 a 10.

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.

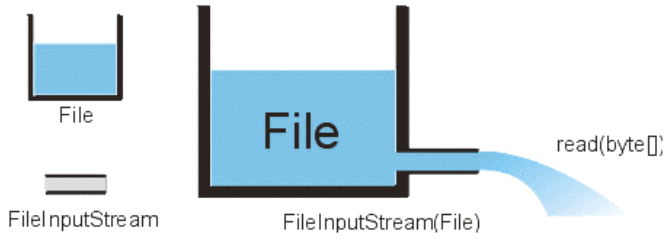


Figura 6-1

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.

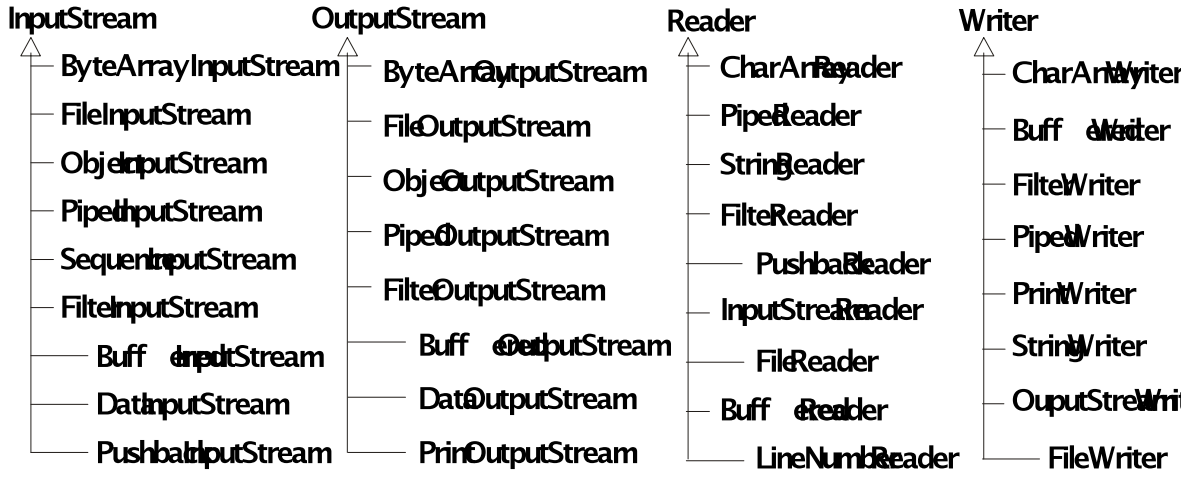


Figura 6-2 – Hierarquia de fluxos de dados e caracteres na plataforma Java 2

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
```

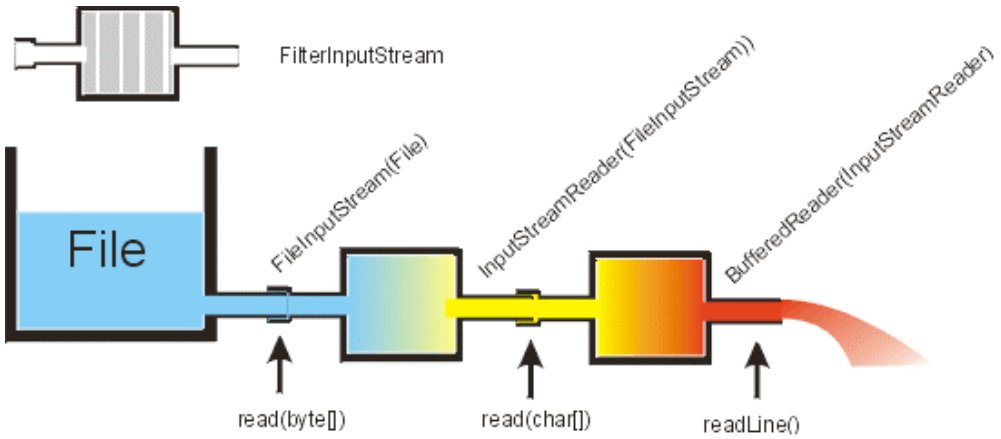


Figura 6-3 – Leitura 'filtrada' a partir de um arquivo

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");
```

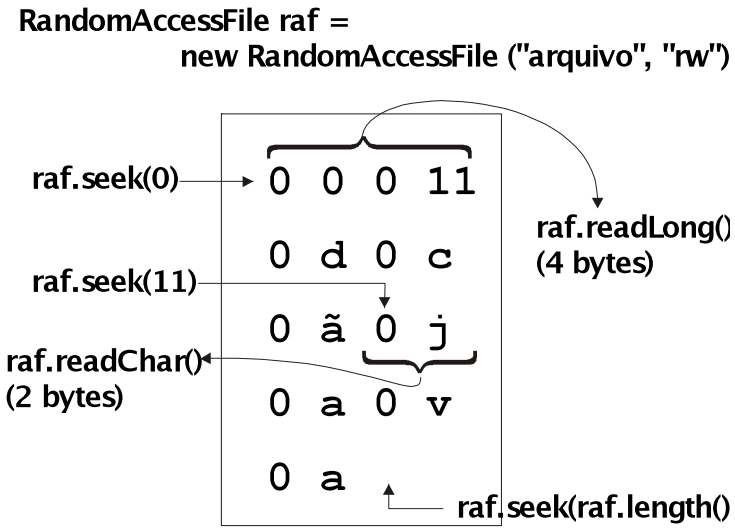


Figura 6-4: RandomAccessFile

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.

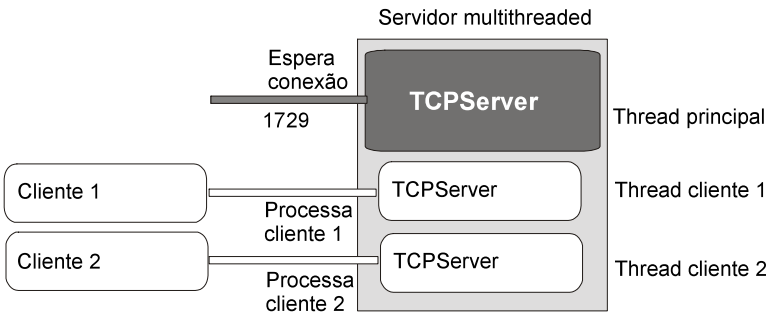


Figura 6-5

`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);
      }
   }
```

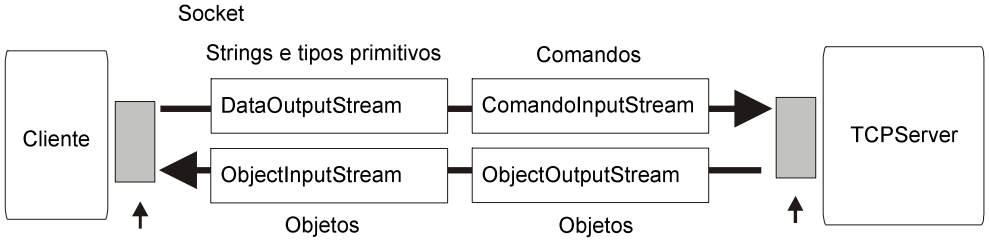


```
(...)
   /** Comando SETREGISTRO */
   public static class SetRegCmd implements Comando {

      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.

