Programação Avançada em Java

de

Patrícia Augustin Jaques

Programação Avançada em Java

Autoria: **Patrícia Augustin Jaques** - *pjaques@unisinos.br*

Última atualização: Outubro de 2007.

Nenhuma parte desta apostila pode ser utilizada ou reproduzida, em qualquer meio ou forma, seja mecânico ou eletrônico, fotocópia, gravação, ou outros, sem autorização, prévia, expressa e específica do Autor. Essa apostila está protegida pela licença Creative Communs. Você pode usá-la para seus estudos, mas não pode usá-la para fins comerciais ou modificá-la. Bons estudos em Java!

"Programação Avançada em Java" by Patrícia Augustin Jaques is licensed under a Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License.

Meus agradecimentos a aluna Natali Silvério que me fez conhecer a licença Creative Commons. Um especial agradecimento ao aluno Leandro Medeiros que ajudou na revisão do texto.

Sumário

1 Tratamento de Exceções

O tratamento de exceções em Java permite o gerenciamento de erros em tempo de execução. Uma exceção em Java é um objeto que descreve uma condição de exceção que ocorreu em algum fragmento de código. Quando surge uma condição excepcional, um objeto *Exception* é criado e lançado no método que causa a exceção.

1.1 Classificação de tipos de Exceções

Em Java, uma exceção é sempre uma instância de uma classe derivada da classe *Throwable*. Veja a figura abaixo:

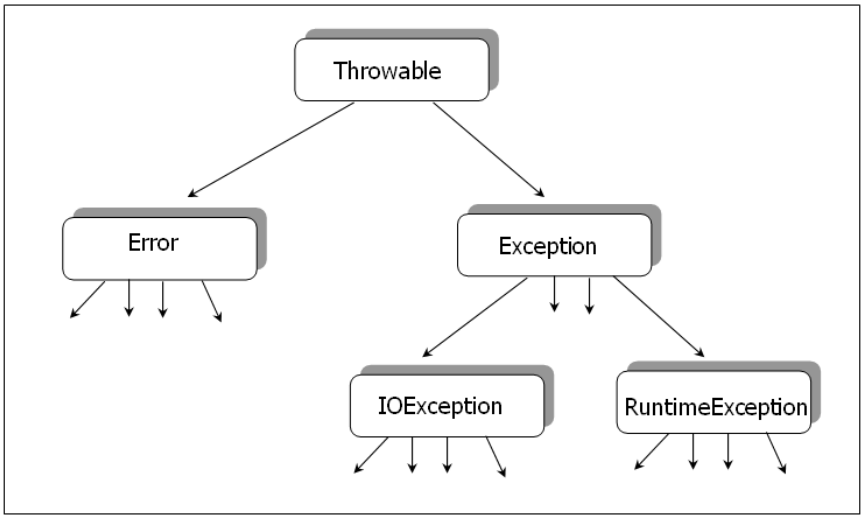

Figura 1: Diagrama simplificado da hierarquia de exceções em Java

Note que todas as exceções são descendentes de Throwable, mas que se dividem em Error e Exception.

A hierarquia de **Error** descreve erros internos e exaustão de recursos dentro do Java Runtime como, por exemplo, estouro de memória. Estes erros não devem ser tratados pelo programador, pois há muito pouco o que se pode fazer quando esses erros acontecem, além de notificar o usuário e finalizar a execução do programa. Esses tipos de erros são mais raros.

O programador deve tratar os erros que descendem do tipo Exception. Eles se dividem em dois: **RuntimeException** e os outros (não RuntimeException). Os erros RuntimeException acontecem porque o código foi mal programado. Os outros erros acontecem devido a alguma coisa que aconteceu de errado no programa, por exemplo, um erro de I/O. As exceções do tipo RuntimeException são criadas de forma automática pelo runtime, em reposta a algum erro do programa.

Exceptions que herdam do RuntimeException inclui problemas tais como:

Um casting mau feito;

Acesso a uma posição não existente de um array;
Acesso a um ponteiro null.

Exceções que não herdam do RuntimeException inclui:

Tentar ler além do final de um arquivo;
Tentar abrir uma URL mal construída;
Erro na formatação de um número inteiro.

Os objetos de exceção são criados automaticamente pelo runtime de Java em resposta a alguma condição de exceção. Por exemplo, o programa abaixo produz uma condição de erro (divisão por zero) em tempo de execução:

```
class ExcecaoDivisaoPorZero {
             public static void main (String args [])  {
                    int d=0;
                    int a=42/d;
                       System.out.println ("Execução continua.");
             }
}
```

Quando o runtime do Java tenta executar a divisão, ele percebe que o denominador é zero e constrói um objeto de exceção para fazer com que a execução do código pare e trate essa execução. Neste exemplo, como não foi descrito nenhum manipulador de exceção (bloco try/catch), o programa pára a execução e é chamado o manipulador de exceção padrão do runtime. O programa acima produz o seguinte resultado:


```
c:\jdk1.3\bin\java.exe   ExcecaoDivisaoPorZero
java.lang.ArithmeticException: / by zero
       at
ExcecaoDivisaoPorZero.main(ExcecaoDivisaoPorZero.java:6)
Exception in thread "main"
```

Observe que são fornecidos o nome da classe, o nome do método, o nome do arquivo e a linha onde ocorreu a exceção.

1.2 try e catch

É sempre mais aconselhável que o programador trate da exceção para que o programa continue a execução. A palavra-chave try pode ser usada para especificar um bloco de código que trata a exceção e catch para especificar qual o tipo de exceção que será capturada. Por exemplo, vamos tratar o programa acima para que a execução não seja finalizada.

```
class ExcecaoDivisaoPorZero {
       public static void main (String args []) {
              try {
                     int d=0;
                     int a=42/d;
                     System.out.println ("Dentro do bloco da exceção.");
              }
              catch (ArithmeticException e) {
                     System.out.println ("Aconteceu divisão por zero.");
              }
              System.out.println ("Execução continua.");
       }
}
```

O código acima produz o seguinte resultado:


```
c:\jdk1.3\bin\java.exe   ExcecaoDivisaoPorZero
Aconteceu divisão por zero.
Execução continua.
```

Quando acontece uma exceção e é encontrado um manipulador de exceção (o bloco try/catch) fornecido pelo programador, o interpretador sai do bloco onde aconteceu a exceção e é executado o manipulador de exceção. Observe que não foi executada a declaração System.out.println ("Dentro do bloco da exceção."). Após, a execução continua a partir da próxima declaração imediatamente após o bloco try/catch.

1.3 Várias Cláusulas Catch

Em alguns casos, pode surgir mais do que uma condição de exceção em uma mesma seqüência de código. O programador pode, então, usar mais de uma cláusula *catch* para tratar esses casos. Cada um dos tipos de exceção é tratado na ordem em que é declarado. Desta maneira, é importante que sempre colocamos a classe mais específica em primeiro lugar. Veja o seguinte exemplo:

```
class MultiCatch {
       public static void main (String args []) {
              try {
                     int a = args.length;
                     System.out.println ("a = "+a);
                     int b = 42/a;
                     int c [ ] = {1};
                     c[42] = 99;
              }
              catch (ArithmeticException e) {
                     System.out.println ("Div por 0: "+e);
              }
              catch (ArrayIndexOutOfBoundsException e) {
                     System.out.println ("Estouro índice array: "+e);
              }
       }
}
```

O exemplo acima gera uma exceção de divisão por zero se o usuário não forneça nenhum parâmetro. Caso o usuário forneça um parâmetro, será gerada uma exceção do tipo ArrayIndexOutOfBoundsException, uma vez que a array c terá comprimento 1 e estamos tentando acessar a posição 42 da array. Os seguintes resultados serão produzidos no exemplo acima, para o primeiro e segundo caso respectivamente.


```
c:\jdk1.3\bin\java.exe   MultiCatch
a = 0
Div por 0: java.lang.ArithmeticException: / by zero
c:\jdk1.3\bin\java MultiCatch 1
a = 1
Estouro índice array: Java.lang.ArrayIndexOutOfBoundsException
```

1.4 Criando a minha classe de exceções

É possível criar o seu próprio tratador de exceções e isto é o tipo mais comum de uso de exceções em Java. Esta classe deve ser derivada da classe Throwable ou de uma de suas subclasses. Estas subclasses não precisam implementar quase nada, mas a sua existência permite módulos de falha mais compreensíveis e uma interface limpa. Veja o exemplo abaixo:

```
class MinhaExcecao extends Exception {
       private int detalhe;

       public MinhaExcecao (int a) {
              detalhe = a;
       }
       public String toString ( ) {
              return "MinhaExcecao ["+detalhe+"]";
       }
} // da class MinhaExcecao
```

1.5 Usando throw para lançar exceções

Uma vez que você crie a sua própria exceção, você precisa lançar essa exceção, ou seja, avisar o interpretador que aconteceu uma exceção que precisa ser tratada. Para tanto, você pode usar a declaração throw. Primeiro você precisa obter uma instância de exceção para a cláusula catch, criando uma instância com o operador new. Veja como fica o código de um programa que lança e trata a exceção MinhaExceção:

```
class DemoExcecao {
       public static void main (String args []) {
              try {
                     int a=11;
                     if (a>10)  {
                            MinhaExcecao minhaExc = new MinhaExcecao (a);
                         throw minhaExc;
                     }
              }
              catch (MinhaExcecao e) {
                     System.out.println ("Excecao capturada: "+e);
              }
       }
} // da class DemoExcecao
```

O fluxo de execução pára imediatamente após o throw e a próxima declaração não é alcançada. A execução é transferida para o bloco em que existe um catch associado para capturar o tipo de exceção lançado.

O throw pode ser usado para lançar exceções criadas pelo usuário ou classes de exceções fornecidas pelo Java. Mas, o throw lança sempre uma instância de exceção e, por isso, essa instância deve ser criada com o operador *new*.

1.5.1 finally

Quando acontece uma exceção, muitas vezes, pode ser necessário que um determinado bloco de código seja executado. Para essa finalidade existe a declaração *finally*. O *finally* garante que um determinado código seja executado, independente de ocorrer uma exceção ou não.

O bloco *finally* será executado antes do código posterior ao bloco try, mesmo que não exista um bloco catch correspondente. Esta cláusula é opcional. Por exemplo, o código abaixo mostra vários métodos que saem de maneiras diferentes, mas apenas após executar o bloco finally.

```
class DemoFinally {
       static void procA () {
              try {
                     System.out.println ("dentro de procA");
                     throw new RuntimeException ("demo");
              }
              finally {
                     System.out.println ("finally de procA");
              }
       }
       static void procB () {
              try {
                     System.out.println ("dentro de procB");
                     return;
              }
              finally {
                     System.out.println ("finally de procB");
              }
       }
       public static void main (String args []) {
              try {
                     procA ();
              }
              catch (Exception e)  {
                     System.out.println ("Tratando exceção que ocorreu no método
procA");
              }
              procB ();
       }
 } // da class
```

O programa acima produzirá o seguinte resultado:


```
c:\jdk1.3\bin\java.exe   DemoFinally
dentro de procA
finally de procA
Tratando exceção que ocorreu no método procA
dentro de procB
finally de procB
```

1.6 throws

Se um método pode causar uma exceção em que ele não trata, ele deve especificar isso, para que os métodos que os chamam possam tratar essa exceção. A palavra-chave throws é usada para especificar a lista de exceções que podem acontecer em um método.

Se a exceção for do tipo Exception ou de uma de suas subclasses, o compilador forçará o programador a especificar essa exceção (com throws ou try/catch). Se o tipo for Error ou RuntimeException ou subclasses destas, não é gerado erro na compilação, uma vez que não se espera que ela ocorra.

É importante lembrar que a cláusula throws apenas lança a exceção um nível acima. Posteriormente, essa exceção deve ser tratada por uma cláusula throw. Veja o exemplo abaixo:

```
class DemoThrows {
       public static void proced () throws MinhaExcecao {
              System.out.println ("No Procedimento.");
              throw new MinhaExcecao (1);
       }
       public static void main (String args []) {
              try {
                     proced ();
              }
              catch (MinhaExcecao e) {
                     System.out.println ("Aconteceu divisão por zero.");
              }
       }
}
```

O programa acima produz o seguinte resultado:


```
c:\jdk1.3\bin\java.exe   DemoThrows
No Procedimento.
Aconteceu divisão por zero.
```

2 Entrada e Saída de Dados em Java

A maioria dos programas precisa acessar e enviar dados externos ao programa. Os dados de entrada podem ser provenientes de um arquivo em disco, de um teclado ou de uma conexão de rede. Da mesma maneira, o destino de saída pode ser um arquivo em disco ou uma conexão em rede. Java permite lidar com todos os tipos de entrada e saída através de uma abstração conhecida como *stream*. Stream é tanto uma fonte de dados como também um destino para dados.

O pacote *java.io* define um grande número de classes para ler e escrever streams. As classes *InputStream* e *OutputStream,* bem como as suas subclasses, são usadas para ler e escrever stream de bytes, enquanto que as classes *Reader* e *Writer* são usadas para ler e escrever streams de caracteres.

As streams são unidirecionais, ou seja, você pode ler de uma stream de entrada (InputStream), mas não pode escrever nela. Da mesma maneira você pode escrever de uma stream de saída (OutputStream), mas não pode ler dela.

Algumas streams são chamadas de "*nodes*" porque elas lêem dados ou escrevem dados de algum lugar em específico como, por exemplo, arquivo de disco ou conexão em rede. Outras streams são chamadas de *filtros,* pois são uma conexão a uma stream de entrada, bufferizando e repassando os dados provenientes desta outra stream de entrada.

Neste capítulo serão abordados diversos exemplos, bem como as classes usadas na execução destes exemplos.

2.1 A Classe InputStream

A classe InputStream fornece métodos para a leitura de bytes a partir de uma stream.

Métodos	Descrição
int read ( )	Lê um byte e retorna o seu valor em um inteiro. Retorna –1 se chegou ao fim do arquivo.
int read (byte b[ ])	Escreve os bytes lidos na array de bytes `b`. Será lido, no máximo, `b.length` bytes. Retorna o número de bytes lidos.
int read (byte b[], int off, int length)	Escreve `length` bytes lidos na array de bytes passada como parâmetro. O primeiro byte lido é armazenado na posição `off` da array. Retorna o número de bytes lidos
void close ( )	Fecha a stream. Se existir uma pilha de stream, fechar a stream do topo da pilha, irá fechar todas as outras streams.
int available ( )	Retorna o número de bytes disponíveis para leitura.
long skip (long nroBytes)	Este método é usado para movimentar o ponteiro do arquivo. Ele descarta `nroBytes` bytes da stream. Retorna o número de bytes descartados.

2.1.1 Lendo a partir do console com a classe InputStream

```
import java.io.*;
public class Console {
  public static void main(String args[]) {
```

```
    str = readString ("Escreva alguma coisa: ");
    System.out.println("Você escreveu: " + str);
  }

  public static String readString(String msg) {
    String str = "";
    char c=' ';
       System.out.print (msg);
    for(;;) {
      try  {
        c = (char)System.in.read();
      } catch(IOException e) {}
      if(c==-1 || c=='\n') break;
      str += c;
    }
    return str;
  }
}
```

2.2 A Classe OutputStream

A classe OutputStream fornece métodos para escrever bytes em uma stream de saída.

Métodos	Descrição
void write (int)	Grava um byte na stream.
void write (byte b [])	Grava os bytes contidos na array b na stream.
void write (byte b[], int off, int length)	Grava length bytes da array para a stream. O byte b[off] é o primeiro a ser gravado.
void flush( )	Algumas vezes a stream de saída pode acumular os bytes antes de gravá-los. O método flush ( ) força a gravação dos bytes.
void close ( )	Fecha a stream.

Tanto a InputStream como a OutputStream são classes abstratas que definem o modelo Java de entrada e saída de stream. Os métodos destas classes são mais usados quando se deseja ler ou escrever um byte de dado.

2.3 Obtendo as propriedades de um arquivo (File)

A classe File é usada para referenciar um arquivo real em disco. Ela é apenas usada para recuperar as informações de um arquivo e não serve para ler e escrever a partir de um arquivo. O File também pode ser usado para acessar diretórios, pois um diretório em Java é um File com uma propriedade adicional: uma lista de nomes de arquivos que pertencem àquele diretório. O exemplo a seguir permite verificar as propriedades de um arquivo:

```
import java.io.File;
import java.util.Date;
import java.text.*;
class TestaArquivo
{
       public static void main (String args [])
       {
              File f = new File ("Circle.java");
              System.out.println ("Nome do arquivo: "+f.getName ());
              System.out.println ("Caminho: "+f.getPath ());
              System.out.println ("Caminho Absoluto: "+f.getAbsolutePath ());
              System.out.println ("Diretório pai: "+f.getParent ());
              System.out.println (f.exists() ? "existe":"não existe");
```

```
            System.out.println (f.canWrite() ? "pode ser gravado":"não pode ser
gravado");
            System.out.println (f.canRead() ? "pode ser lido":"não pode ser
lido");
            System.out.println (f.isDirectory () ? "é diretório":"não é
diretório");
            DateFormat df = new SimpleDateFormat ("dd/MM/yyyy");
            Date data = new Date (f.lastModified ());
            System.out.println ("Ultima modificação do arquivo: "+df.format
(data));
            System.out.println ("Tamanho do arquivo: "+f.length ()+ " bytes.");
      }
} // da class TestaArquivo
```

O programa acima gera o seguinte resultado:


```
c:\jdk1.3\bin\java.exe   TestaArquivo
Nome do arquivo: Circle.java
Caminho: Circle.java
Caminho Absoluto:
C:\Cursos\CursoJava\MeusExercicios\Circle.java
Diretório pai: null
existe
pode ser gravado
pode ser lido
não é diretório
Ultima modificação do arquivo: 09/05/2000
Tamanho do arquivo: 358 bytes.
```

É importante lembrar que a classe File não cria arquivos não existentes, ela apenas referencia arquivos existentes. Para criar um novo arquivo, devem ser usadas as streams (classes FileWriter e FileOutputStream).

2.4 Listando diretórios

Um diretório é um File contendo uma lista de outros arquivos e diretórios. É possível saber se um objeto File é um diretório pelo método isDirectory. Pode-se, então, usar o método list para listar arquivos e diretórios contidos neste diretório. Se este método for usado para um objeto que não é diretório, será gerado um NullPointerException já que ele não criará uma array de String para o método list( ) retornar.

O programa abaixo imprime a lista de arquivos e diretórios de um diretório:

```
import java.io.File;
class ListaDir {
      public static void main (String args []) {
            String nomeDir = "c:/";
            File f1 = new File (nomeDir);
            if (f1.isDirectory ( )) {
                  System.out.println ("Diretório "+nomeDir);
                  String s[] = f1.list( );
                  for (int i=0; i<s.length; i++) {
                        File f = new File (nomeDir + "/" + s[i]);
                        System.out.print (s[i]);
                        if (f.isDirectory ( )) System.out.println (" <dir> ");
                        else System.out.println (" <file>");
                  }
            }
            else System.out.println (nomeDir + " não é um diretório.");
      }
} // da class
```

O programa acima gera a seguinte saída:


```
c:\jdk1.3\bin\java.exe   ListaDir
Diretório .
Minhas Webs <dir>
desktop.ini <file>
Minhas imagens <dir>
ListaDir.java <file>
ListaDir.class <file>
```

2.5 FileInputStream e FileOutputStream

Estas classes são "node" streams e são usadas para ler de um arquivo em disco (FileInputStream) e para escrever em um arquivo em disco (FileOutputStream). Elas dispõem dos mesmos métodos fornecidos nas classes InputStream e OutputStream.

```
import java.io.*;
public class CopiaArquivo {
  public static void main(String arg[]) throws IOException {
    FileInputStream in = new FileInputStream("CopiaArquivo.java");
    FileOutputStream out = new FileOutputStream("Copia de CopiaArquivo.java");
    int c;
    while ((c = in.read()) != -1)
       out.write(c);
    in.close();
    out.close();
  }
}
```

2.6 A Classe BufferedReader

2.6.1 Utilizando a classe BufferedReader para ler do console

```
import java.io.*;

public class Console {
  public static void main(String args[]) {
    String str = read("Escreva alguma coisa: ");
    System.out.println("Voce escreveu: " + str);
}
  public static String read (String str) {
             InputStream in = System.in;
             InputStreamReader is = new InputStreamReader (in);
             BufferedReader console  = new BufferedReader (is);
             System.out.print (str);
             String name = null;
             try {
                    name = console.readLine ();
             }
             catch (IOException e) { }
          return name;
  }
}
```

Os dados de entrada são obtidos através de uma InputStream, variável estática da classe System. Para fazer a leitura de linhas, usamos o método readLine( ) da classe BufferedReader. Uma instância da classe InputStreamReader é criada como interface entre a classe InputStream e a classe BufferedReader (subclasse de Reader).

2.6.2 Lendo linhas de um arquivo

Para ler linhas de um arquivo, podemos igualmente usar os métodos da classe BufferedReader. O código abaixo lê as linhas de um arquivo e finaliza a execução após o final do arquivo:

```
import java.io.*;
class LeArquivo {
	public static void main (String args []) {
		String filename = "Circle.java";
		try {
			FileReader fr = new FileReader (filename);
			BufferedReader in = new BufferedReader (fr);

			String line;
			while ( (line=in.readLine())!= null) System.out.println (line);
			in.close ();
		}
		catch (IOException e) {
			System.out.println ("Erro na leitura do arquivo "+filename);
		}
	}
} // da class
```

O programa acima imprime no console o arquivo lido "Circle.java".

2.7 Escrevendo linhas em um arquivo

Quando desejamos armazenar em um arquivo um conteúdo String, por exemplo, linhas de texto, usamos os métodos da classe PrintWriter.

```
import java.io.*;
class EscreveArquivo {
	public static void main (String args [ ]) {
		try {
			File f = new File ("MeuArquivo.txt");
			FileWriter fr = new FileWriter (f);
			PrintWriter out = new PrintWriter (fr);
			out.println ("Curso de Java: Arquivo gerado pelo programa.");
			out.close();
		}
		catch (IOException e) {
			System.out.println ("Erro ao escrever arquivo.");
		}
	}
} // da class
```

O código acima cria um arquivo chamado "MeuArquivo.txt" no diretório corrente com o seguinte conteúdo: "Curso de Java: Arquivo gerado pelo programa".

Lembre que FileReader e FileWriter lêem e escrevem caracteres de 16 bits (Unicode). Entretanto, a maioria dos sistemas de arquivos nativos está baseada em 8-bits. Estas streams realizam a conversão para ler e escrever bytes de acordo com a codificação usada pela plataforma. Por exemplo, ao utilizar o FileReader para ler um arquivo texto (txt) , ele lê cada byte (caracter ASCII) e o converte para um caracter Unicode (codificação usada pelo Java). Você pode saber a codificação default da plataforma em que seu programa Java está executando usando o método System.getProperty("file.encoding"). Para especificar uma codificação que não a default, você deve

construir um OutputStreamWriter em uma FileOutputStream e especificá-lo. Para maiores informações sobre codificação de caracteres, veja o tutorial http://java.sun.com/docs/books/tutorial/i18n/index.html.

2.8 DataInputStream e DataOutputStream

Esta stream de filtro permite ler e escrever tipos primitivos de dados através de streams. São fornecidos vários métodos para os diferentes tipos de dados.

DataInputStream	
byte readByte ( )	Lê um byte.
long readLong ( )	Lê um dado long (8 bytes).
double readDouble ( )	Lê um dado double (8 bytes).
char readChar ( )	Lê um caracter (2 bytes).
char readFloat ( )	Lê um dado float (4 bytes).
int readInt ( )	Lê um inteiro (4 bytes).

DataOutputStream	
void writeByte (byte)	Grava um byte.
void writeLong (long)	Grava um dado long (8 bytes).
void writeDouble (double)	Grava um dado double (8 bytes).
void writeChar (char)	Grava um caracter (2 bytes).
void writeFloat (char)	Grava um dado float (4 bytes).
void writeInt (int)	Grava um inteiro (4 bytes).

```
import java.io.*;
public class EscreveStream {
  public static void main(String arg[]) throws IOException {
    File f=new File("dados.dat");
    FileOutputStream fos=new FileOutputStream(f);
    DataOutputStream dos=new DataOutputStream(fos);
    int n=5;
    float fo=3.14f;
    char c='A';
    dos.writeInt(n);
    dos.writeFloat(fo);
    dos.writeChar(c);
// Apenas precisa fechar a stream mais alta. As demais são fechadas automaticamente.
    dos.close();
  }
}
```

```
public class UsaDataInputStream {
  public static void main(String arg[]) throws IOException {
    File f=new File("dados.dat");
    FileInputStream fos=new FileInputStream(f);
    DataInputStream dos=new DataInputStream(fos);
    int n = dos.readInt();
    float flo = dos.readFloat();
    char c = dos.readChar();
    System.out.println ("Foram lidos os valores: ");
    System.out.println ("(int) = "+n);
```

```
    System.out.println ("(float) = "+flo);
    System.out.println ("(char) = "+c);
    dos.close();
  }
}
```

Se você gravar em um arquivo tipos diferentes de dados primitivos (int, float, char, etc), você deve lê-los na mesma ordem em que escreveu.

2.8.1 Lendo todo um arquivo para um array de bytes

No caso de você estar construindo um editor de texto, você pode muitas vezes querer mostrar todo um texto de uma vez na tela ao usuário. Para tanto, é interessante usar um método que leia todo o texto para a memória ao invés de ler bytes ou linhas. O programa abaixo demonstra como fazer isso:

```
import java.io.*;
class LeTodoArquivo {
      public static void main (String args []) {
            String filename = "Circle.java";
            try {
                  File f = new File (filename);
                  FileInputStream fis = new FileInputStream (f);
                  DataInputStream dis = new DataInputStream (fis);

                  int tamArq = (int) f.length();
                  byte [] arq = new byte [tamArq];
                  dis.readFully (arq);
                  System.out.print (new String(arq));
                  dis.close ();
            }
            catch (IOException e) {
                  System.out.println ("Erro na leitura do arquivo "+filename);
            }
      }
} // da class
```

O programa acima lê todo o conteúdo de um arquivo para um array de bytes, transforma em uma string e imprime o conteúdo na tela.

Você também poderia usar o método write (byte[ ] b) da classe FileOutputStream para imprimir todo o conteúdo da array de bytes para um outro arquivo (duplicação do arquivo).

2.9 Armazenando objetos em arquivos

Uma importante característica do pacote *java.io* é a capacidade de serializar objetos, ou seja, converter um objeto em uma stream de bytes que depois podem ser convertidas novamente em um objeto. Esta capacidade nos permite armazenar objetos em arquivos e recuperá-los do arquivo para um objeto.

Para que um objeto possa ser serializado, ele tem que implementar a classe *Serializable*. É importante lembrar que apenas as variáveis são armazenadas, os métodos não. O objeto serializável pode conter tantas variáveis do tipo básico (boolean, int, char) como também referência a objetos.

Todos são igualmente armazenados. Porém, os objetos referenciados também devem ser Serializable. Além disso, se é desejável que uma variável não seja armazenada, ela pode ser marcada como *transient*, Para tanto, é preciso apenas colocar o modificador *transient* na declaração da variável. Isso é bastante útil quando o objeto que se deseja armazenar contém uma referência para um outro objeto que não é serializável. As variáveis estáticas também não são armazenadas.

O exemplo abaixo apresenta a serialização do objeto Data.

```
import java.io.*;
class Serializacao {
	public static void main (String args [ ]) {
		Pessoa p = new Pessoa ("Homem Aranha", 5, 8, 1937);
		File f = new File ("ArqSerializacao.arq");
		gravaObjeto (f, p);
		Pessoa p2 = (Pessoa) leObjeto (f);
		System.out.println ("Foi armazendo o objeto pessoa com os valores:");
		System.out.println ("Nome: "+p2.getNome()+" \nData: "+p2.getData());
	}
	private static void gravaObjeto (File f, Object o) {
		try {
			FileOutputStream fos = new FileOutputStream (f);
			ObjectOutputStream os = new ObjectOutputStream (fos);
			os.writeObject (o);
			os.close ();
		}
		catch (IOException e)  {
			System.out.println ("Erro ao gravar objeto.");
		}
	}
	private static Object leObjeto (File f) {
		Object o = null;
		try {
			FileInputStream fos = new FileInputStream (f);
			ObjectInputStream os = new ObjectInputStream (fos);
			o = os.readObject ();
			os.close ();
		}
		catch (IOException e)  {
			System.out.println ("Erro ao abrir arquivo.");
		}
		catch (ClassNotFoundException ce) {
			System.out.println ("Objeto não encontrado.");
		}
		return o;
	}
} // da class Serializacao
/* ************************************ */
class Pessoa implements Serializable {
	String nome;
	Data d;

	public Pessoa (String nome, int dia, int mes, int ano) {
		this.nome = nome;
		d = new Data (dia, mes, ano);
	}
	public String getNome () {
		return nome;
	}
	public String getData () {
		return d.getData ();
	}
} // da class Pessoa
/* ************************************ */
```

```
class Data implements Serializable {
	int dia;
	int mes;
	int ano;
	public Data (int dia, int mes, int ano) {
		this.dia = dia;
		this.mes = mes;
		this.ano = ano;
	}
	public String getData () {
		return dia+"/"+mes+"/"+ano;
	}
} // da class Data
```

O programa acima gera a seguinte saída:


```
c:\jdk1.3\bin\java.exe   Serializacao
Foi armazendo o objeto pessoa com os valores:
Nome: Homem Aranha
Data: 5/8/1937
```

2.10 Arquivos de acesso randômico

Você deve ter percebido, que até agora, para acessar certos dados de um arquivo nós tínhamos que lê-lo do início ao fim. Você pode desejar ler dados de um arquivo diretamente em alguma parte dele, sem ter que ler o arquivo do início até o fim. Para esse fim, você vai usar a classe RandomAccessFile.

Além disso, o RandomAccessFile permite abrir um arquivo para leitura e escrita ou apenas para leitura. Veja o exemplo abaixo:

```
import java.io.*;
class ArquivoAcessoRandomico {
	public static void main (String args []) throws java.io.IOException {
		RandomAccessFile raf = new RandomAccessFile ("Raf.arq", "rw");
		raf.writeBytes ("Escrevendo a primeira linha.\n");
		raf.writeBytes ("Escrevendo a segunda linha.\n");
		raf.seek (0);
		String s = raf.readLine ();
		System.out.println ("Primeira linha no texto: "+s);
		raf.seek (raf.length()); // vai para o final do arquivo
		raf.writeBytes ("Escrevendo a última linha.\n");
	}
}
```

O programa acima grava duas strings (seguidas de nova linha), lê o arquivo a partir da posição zero até encontrar o caracter de nova linha (uma linha de texto) e após move o ponteiro para o final do arquivo, para gravar outra string. Vale lembrar que se o ponteiro do arquivo não fosse movido para a última posição (como estava na posição zero), o novo texto, ao invés de ser copiado para o final do arquivo, sobrescreveria o texto já gravado no arquivo.

3 Threads

Programação Multithread é um paradigma conceitual da programação pelo qual você divide os programas em dois ou mais processos que podem ser executados paralelamente.

Muitas vezes, o programa implementado não exige os recursos completos do computador. Por exemplo, o computador pode levar um minuto para ler os dados do usuário do teclado, mas o tempo em que a CPU estará envolvida nesse processo é mínimo. Devido a isso, a CPU fica ociosa grande parte do seu tempo. O problema dos ambientes tradicionais de linha de execução única é que você precisa esperar que cada uma dessas tarefas termine para após passar para a outra, ao invés de aproveitar o tempo ocioso da CPU para executar outra tarefas.

Além disso, as *threads* podem se comunicar e compartilhar estruturas de dados, o que pode ser muito útil.

Para determinar que certas classes do seu programa podem executar paralelamente, você tem de defini-las como uma thread. Podemos definir uma classe como Thread de duas maneiras: implementando a interface Runnable ou derivando da classe Thread.

3.1 Criando uma Thread

No exemplo abaixo, temos um exemplo de uma Thread.

```
class MinhaThread1 extends Thread {
	public MinhaThread1 (String nome) {
		super (nome);
	}
	public void run () {
		for (int i=0; i<4; i++) {
			System.out.println (getName() + " " +i);
			try {
				sleep (400);
			}
			catch (InterruptedException e) { }
		}
	}
	public static void main (String args[]) {
		MinhaThread1 t1 = new MinhaThread1 ("Dançando");
		MinhaThread1 t2 = new MinhaThread1 ("Assobiando");
		t1.start ();
		t2.start ();
	}
}
```

O programa acima produz o seguinte resultado:


```
c:>java MinhaThread1
Dançando 0
Assobiando 0
Dançando 1
Assobiando 1
Dançando 2
Assobiando 2
Dançando 3
Assobiando 3
```

Toda thread deve ter um método run(). É neste método que é iniciada a execução da thread. Podemos pensar no método run() como o equivalente para threads do método main(). Uma thread inicia a sua execução pelo método run(), assim como uma aplicação Java inicia a sua execução pelo método main().

O método run() não aceita parâmetros. Por isso, quando você desejar passar parâmetros para uma thread, pode fazer isso através do método construtor.

A chamada do método start() da classe Thread, cria uma nova thread e executa o método run() definido nesta classe thread.

O método sleep() é um método estático da classe Thread que põe a thread corrente para dormir durante um número de milisegundos especificado. Em Java, a thread precisa falar para outra thread que ela está inativa, então outras threads podem pegar a chance de executar. Uma maneira de fazer isso é através do método sleep(). No exemplo acima, a classe MinhaThread1 usou o método sleep() para avisar que iria ficar inativa durante 400 milisegundos e que, durante este tempo, outras threads poderiam ser executadas. Você pode observar que se retirássemos a chamada do método sleep() do código, seria executado primeiro todo o laço for da primeira thread para após ser executado o laço for da outra thread. Isso porque a primeira thread não ficou ociosa para a segunda thread poder executar. Sem o método sleep() dentro do laço for, o programa acima geraria a seguinte saída:


```
c:\> java   MinhaThread1
Dancando 0
Dancando 1
Dancando 2
Dancando 3
Assobiando 0
Assobiando 1
Assobiando 2
Assobiando 3
Dancando
Dancando
Dancando
Dancando
Assobiando
Assobiando
Assobiando
Assobiando
```

Da mesma maneira, é possível criar a thread acima, implementando a interface Runnable. Como Java não aceita herança múltipla, se a classe thread já for filha de outra classe, não será possível fazer com que ela herde da classe da Thread também. Nestes casos, ao invés de herdar da classe Thread, a classe pode implementar a interface Runnable.

```
class MinhaThread2 implements Runnable {
       String nome;
       public MinhaThread2 (String nome) {
              this.nome = nome;
       }
```

```
        public void run () {
                for (int i=0; i<4; i++) {
                        System.out.println (nome+" "+i);
                        try {
                                Thread.sleep (400);
                        }
                        catch (InterruptedException e) { }
                }
        }
        public static void main (String args[]) {
                MinhaThread2 c = new MinhaThread2 ("Dançando");
                MinhaThread2 c2 = new MinhaThread2 ("Pulando");
                Thread t1 = new Thread (c);
                Thread t2 = new Thread (c2);
                t1.start ();
                t2.start ();
        }
}
```

O programa acima produz resultado idêntico ao anterior.

3.2 Estados das Threads

As threads podem estar em um destes estados (ver Figura 2):

New: Uma thread está no estado de nova quando ela foi criada pelo operador new. Neste estado, Java ainda não está executando o código dentro dela.

Runnable: Quando o método start() da thread é invocado, ela passa para o estado de runnable. Quando o código dentro da thread começa a ser executado, ela está executando (ou running).

Blocked: Quando uma thread entra no estado de bloqueado, uma outra thread é escalonada para executar. Quando ela é reativada (por exemplo, porque ela já acabou o tempo de sleep), o escalonador verifica se ela tem prioridade maior que a thread que está executando. Em caso positivo, ele preempta a thread corrente e começa a executar a thread bloqueada novamente.

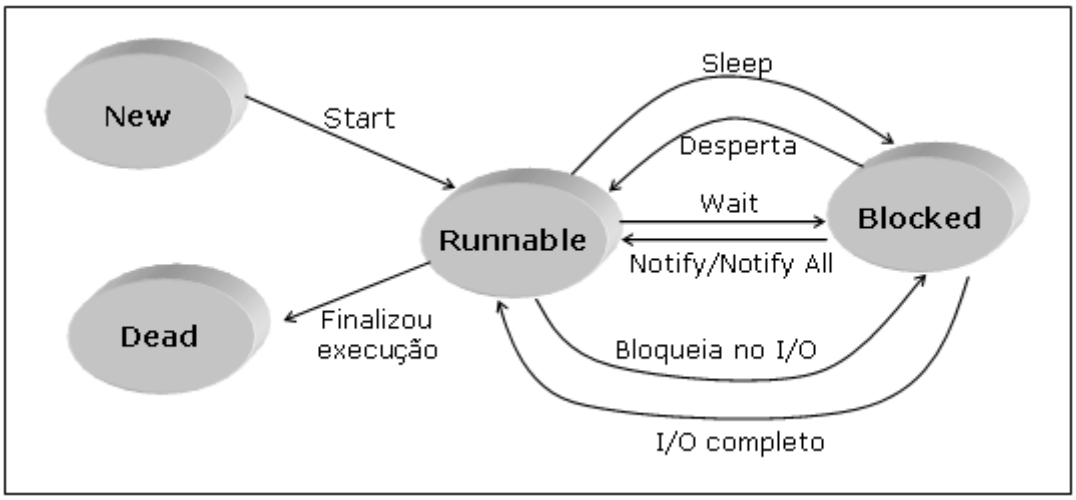

Figura 2: Estados possíveis de uma thread

Uma thread entra em um estado de bloqueado, quando uma das ações ocorre:

É chamado o método sleep() da thread;
A thread chama o método wait();
A thread está bloqueada em uma operação de I/O, por exemplo, esperando o usuário digitar alguma coisa no teclado.
Dead: Uma thread está morta quando ela parou a sua execução.

Para saber o estado de uma thread podemos usar o método isAlive(). Este método retorna true, caso a thread esteja runnable ou bloqueada.

3.3 Prioridade de Threads

As prioridades de threads são usadas para definir quando cada thread deve ter permissão para ser executada. Quando uma thread de prioridade mais alta torna-se ativa ou sai de uma espera de I/O, ela deve poder executar imediatamente, com preempção da thread de prioridade mais baixa. As threads de mesma prioridade devem fazer a preempção mútua através de um algoritmo de escalonamento "round-robin" para compartilhar o tempo de CPU.

O pacote de Threads em Java precisa trabalhar com o Sistema Operacional. No sistema Solaris, uma thread executando é mantida até que uma thread de prioridade mais alta tome o controle. Os sistemas, como Windows 95 e Windows NT, dão para cada thread runnable uma fatia de tempo para executar (time-slicing). Após essa fatia de tempo, o Sistema Operacional dá a outra thread a chance de executar.

Cada thread em Java tem uma prioridade. É possível determinar a prioridade de uma thread com o método *setPriotity(int prioridade)*. Os valores para a prioridade podem variar de 1 a 10. A prioridade default é 5.

A JVM possui um mecanismo para escalonar threads chamado escalonador. Este mecanismo tenta escalonar a thread de maior prioridade que esteja no estado runnable. Essa thread permanecerá executando até que:

É chamado o método yield();
A thread finalize a sua execução;
Uma thread de maior prioridade torna-se runnable.

Se um dos casos acima acontecer, uma nova thread será escalonada.

Para garantir que uma thread fique inativa, para que outras threads possam ocupar a CPU, pode ser usado o método yield(). Por exemplo, uma thread pode chamar o método yield() ou sleep() sempre que estiver dentro de um grande laço for para não monopolizar o sistema. Threads que não seguem essa regra são chamadas egoístas.

Haverá cada prioridade, uma fila de threads com a mesma prioridade (10 filas). Se outras threads de mesma prioridade estão runnable, yield() põe a thread que está executando atualmente na última posição da sua fila e chama outra thread para começar. Se não existem outras threads com mesma prioridade, o yield() não faz nada.

Note que a chamada do método sleep() permite que outras threads de prioridade menor possam executar. Porém, o método yield() apenas dá a chance de executar para threads de mesma ou maior prioridade.

```
class ThreadsPrioridade extends Thread {
       public ThreadsPrioridade (String nome) {
              super (nome);
       }
       public ThreadsPrioridade (String nome, int prioridade) {
              super (nome);
              setPriority (prioridade);
       }
       public void run () {
              for (int i=0; i<4; i++) {
                     System.out.println (getName() + " " +i);
                     yield();
              }
       }
       public static void main (String args[]) {
              ThreadsPrioridade t1 = new ThreadsPrioridade ("Menor", 3);
              ThreadsPrioridade t2 = new ThreadsPrioridade ("Maior", 7);
              ThreadsPrioridade t3 = new ThreadsPrioridade ("Default1");
              ThreadsPrioridade t4 = new ThreadsPrioridade ("Default2");
              t1.start ();
              t2.start ();
              t3.start ();
              t4.start ();
       }
}
```

O programa acima gera a seguinte saída:


```
c:\>java ThreadsPrioridade
Maior 0
Maior 1
Maior 2
Maior 3
Default1 0
Default2 0
Default1 1
Default2 1
Default1 2
Default2 2
Default1 3
Default2 3
Menor 0
Menor 1
Menor 2
Menor 3
```

Note que mesmo que tenhamos criado a thread menor primeiramente, ela foi a última a ser executada por possuir menor prioridade. Como dentro do laço for há um método yield(), as threads Default1 e Default2 que têm prioridades iguais, executam e passam a vez para outra thread sucessivamente a cada iteração do laço. Se não houvesse o comando yield(), a thread Default primeiro executaria até finalizar para após chamar a outra thread. Sem o yield(), o programa produziria o seguinte resultado:


```
c:\>java ThreadsPrioridade
Maior 0
Maior 1
Maior 2
Maior 3
Default1 0
Default1 1
Default1 2
Default1 3
Default2 0
Default2 1
Default2 2
Default2 3
Menor 0
Menor 1
Menor 2
Menor 3
```

3.4 Sincronização

Uma thread executando pode acessar qualquer objeto a qual ela tem referência. Desta maneira, é possível que duas threads tentem acessar o mesmo objeto, interferindo umas nas outras. Deve-se tomar o cuidado que as threads acessem o objeto uma por vez. Isto é conhecido como o problema da sincronização.

Para lidar com esse problema, Java expõe o conceito de monitor. Todo o objeto possui um monitor implícito. Quando uma thread tenta acessar um método marcado como *synchronized* de um determinado objeto, ele entra no monitor deste objeto. Todas as outras threads que desejarem acessar um método *synchronized* deste objeto devem esperar. Para a thread liberar o monitor para que outra thread possa acessar o objeto, ela precisa apenas sair do método *synchronized*.

Para entender melhor, vejamos um exemplo. O programa a seguir cria três threads que acessam um mesmo objeto alvo.

```
class SemSincronismo  {
       public static void main (String args[]) {
              Chamame alvo = new Chamame();
              Chamador t1 = new Chamador (alvo, "Oi");
              Chamador t2 = new Chamador (alvo, "Mundo");
              Chamador t3 = new Chamador (alvo, "Sincronizado!");
              t1.start ();
              t2.start ();
              t3.start ();
       }
 }
class Chamador extends Thread {
       Chamame alvo;
       String msg;
       public Chamador (Chamame alvo, String s) {
              this.alvo = alvo;
              msg = s;
       }
       public void run () {
              alvo.chamar (msg);
       }
}
class Chamame {
       public void chamar (String msg) {
              System.out.print ("[" + msg);
              try  {
                     Thread.sleep (1000);
```

```
            }catch (InterruptedException e) {};
            System.out.println ("]");
      }
}
```

O programa acima gera a seguinte saída:


```
c:\jdk1.3\bin\java.exe   SemSincronismo
[Oi[Mundo[Sincronizado!]
]
]
```

Esse resultado obtido se deve porque não existe nada determinando que se quer que aquele pedaço de código seja atômico (indivisível). Neste caso, podemos usar a palavra-chave synchronized para garantir que as threads não executem aquele método ao mesmo tempo.

No exemplo acima é necessário apenas modificar o método chamar da classe Chamame:

```
class Chamame {
      public synchronized void chamar (String msg) {
            System.out.print ("[" + msg);
            try  {
                  Thread.sleep (1000);
            }catch (InterruptedException e) {};
            System.out.println ("]");
      }
}
```

E então, seria gerada a saída desejada:


```
c:\>java   SemSincronismo
[Oi]
[Mundo]
[Sincronizado!]
```

3.5 Interação entre Threads – wait() e notify()

Em alguns casos, pode ser mais interessante algum evento gerar a execução da thread do que ela ter que verificar repetidamente uma condição, o que geraria desperdício da CPU. Por exemplo, vejamos o clássico problema de fila do produtor e consumidor em que o consumidor deve esperar o produtor gerar alguma coisa para poder consumir.

Nesses casos, podemos usar os métodos wait(), notify e notifyAll (). Esses métodos devem ser chamados apenas dentro de métodos synchronized e servem para:

wait( ): diz a thread atual para desistir do monitor e ficar inativa até que outra thread entre no mesmo monitor e chame notify().
notify( ): torna ativa a primeira thread que chamou o método wait();
notifyAll( ): torna ativa todas as threads que chamaram wait no mesmo objeto.

O programa a seguir, é formado por três classes. A classe Fila representa uma fila de elementos (array). O produtor é responsável por adicionar elementos na fila e o Consumidor por

retirar elementos da fila. Quando a fila estiver vazia (tamanho igual a zero), o método wait() é chamado e o consumidor fica inativo a espera de que seja ativado com o método notify(). Isso acontece quando o Produtor coloca mais elementos na fila.

```
class ProdutorConsumidor {
	public static void main (String args []) {
		Fila fila = new Fila ();
		Produtor produtor = new Produtor (fila);
		Consumidor consumidor = new Consumidor (fila);
		produtor.start ();
		consumidor.start ();
	}
}
class Produtor extends Thread {
	Fila fila;
	public Produtor (Fila fila) {
		this.fila = fila;
	}
	public void run () {
		char c;
		for (int i=0; i<20; i++)  {
			c = (char)('a'+i);
			fila.push (c);
			System.out.println ("Produziu: "+c);
			try {
				Thread.sleep ((int)(Math.random()*100));
			}
			catch (InterruptedException e) { }
		}
     }
}
class Consumidor extends Thread {
	Fila fila;
	public Consumidor (Fila fila) {
		this.fila = fila;
	}
	public void run () {
		char c;
		for (int i=0; i<20; i++)  {
			c = fila.pop ();
			System.out.println ("Consumiu: "+c);
			try {
				Thread.sleep ((int)(Math.random()*100));
			}
			catch (InterruptedException e) { }
		}
	}
}
class Fila {
	private int index=0;
	private char [] buffer = new char [6];
	public synchronized char pop () {
		while (index==0)  {
			System.out.println ("Mandou esperar.");
			try {
				wait();
			} catch (InterruptedException e) { }
		}
		index--;
		return buffer[index];
	}
	public synchronized void push (char c) {
		buffer[index] = c;
		index++;
```

```
            notify();
        }
}
```

O programa acima produz o seguinte resultado:


```
c:\> java   ProdutorConsumidor
Consumiu: a
Produziu: b
Consumiu: b
Mandou esperar.
Produziu: c
Consumiu: c
Mandou esperar.
Produziu: d
Consumiu: d
Produziu: e
Consumiu: e
Produziu: f
Consumiu: f
Mandou esperar.
...
```

No programa acima, se não tivéssemos chamado o método wait() no método pop() do objeto Fila, o Consumidor tentaria acessar a posição –1 do array e seria gerada uma Exception.

4 Redes com java

No pacote java.net você encontra as classes específicas para trabalhar com redes usando a linguagem Java, são eles:URL, Socket, etc. Neste capítulo, nós abordaremos o uso de redes através de sockets Java.

4.1 Programação Cliente-Servidor com Sockets

Os sockets TCP/IP, são usados para implementar conexões confiáveis, bidirecionais e ponto a ponto, com base em um stream entre computadores. Os sockets podem ser usados para conectar o sistema de E/S de um programa Java a outros programas em outras máquinas na Internet.

Para estabelecer uma conexão via socket entre duas máquinas é necessário que uma máquina contenha um programa que fique esperando por uma conexão e a outra tenha um programa que tente se conectar a primeira.

Para tanto, existem duas classes principais:

ServerSocket: Fica ouvindo a porta à espera de que um cliente se conecte a ele.
Socket: Socket cliente que se conecta a um servidor para comunicação.

Assim como para fazer uma ligação precisamos do número de telefone de uma pessoa, para um programa se conectar a uma máquina remota, ele precisa saber o endereço na Internet daquele programa que é o endereço IP. Além disso, é necessário também especificar o número de porta do programa servidor.

Os números de porta em TCP/IP são números de 16 bits que variam de 0-65535. Na prática, portas com número menor que 1024 são reservadas para serviços pré-definidos, tais como FTP, telnet e http e, por isso, não devem ser usadas.

4.1.1 Criando um servidor

```
import java.net.*;
import java.io.*;
public class Server {
  public static void main (String args []) {
    Socket client;
    ServerSocket server;
    try {
              server = new ServerSocket (5000);
            while (true) {
                      System.out.println ("Esperando conexão de cliente!");

                    client = server.accept();
                      System.out.println ("Cliente se conectando.");
                      InputStream in = client.getInputStream (); // obtendo stream de
entrada
                      BufferedReader entrada = new BufferedReader (new
InputStreamReader (in));
                      boolean done = false;
                      while (!done) {
                              String str = entrada.readLine();
                             if (str==null || str.equals("BYE"))  {
```

```
                                    done = true; // conexão foi finalizada pelo cliente
                                    System.out.println ("Conexão finalizada.");
                              }
                              else System.out.println ("Recebeu: "+str);
                        }
                        client.close ();
                  }
      }
      catch (IOException ioe) { System.out.println ("Erro ao aceitar conexao cliente.
Erro: "+ioe); }
     }
} // class
```

O construtor ServerSocket cria um servidor que fica esperando por conexão de um cliente. Quando um cliente requisita uma conexão, o servidor abre um socket com o método accept().

O servidor se comunica com o cliente usando InputStream (recebe dados) e OutputStream (envia dados).

O servidor fica continuamente lendo as mensagens do cliente até que a execução seja finalizada. Neste caso, ele fica esperando a conexão de um novo cliente. Para finalizar o servidor é necessário teclar CTRL+C, já que ele está preso em um laço infinito a espera de novas conexões.

4.1.2 Criando um programa Cliente

Um programa cliente que se conecta a esse servidor, poderia ter o seguinte código:

```
import java.net.*;
import java.io.*;
public class Cliente {
  public static final int PORTA = 5000; //numero da porta do servidor

  public static void main(String args[]) throws IOException {
    Socket s1 = new Socket("127.0.0.1", PORTA);
      OutputStream out = s1.getOutputStream (); // obtendo outputstream
      PrintStream saida = new PrintStream (out);
      boolean done = false;
      while (!done) {
            String str = Console.read (": ");
            saida.println (str);
            if (str.equals ("BYE")) done=true;
      }
      s1.close();
  }
}
class Console {
      public static String read (String str) {
            InputStream in = System.in;
            InputStreamReader is = new InputStreamReader (in);
            BufferedReader console  = new BufferedReader (is);
            System.out.print (str);
            String name = null;
            try {
                  name = console.readLine ();
            }
            catch (IOException e) { }
          return name;
      }
} // da class
```

O programa acima se conecta a um servidor em um determinado endereço e porta. Após isso, fica lendo as mensagens do usuário e enviando ao servidor. Ele fará isso até que o usuário digite "BYE".

4.1.3 Trabalhando com Múltiplos Clientes

Até agora, o servidor aceita a conexão de apenas um cliente por vez. Em alguns casos (por exemplo, um programa de Chat) pode ser útil que o servidor possa aceitar a conexão de mais de um cliente ao mesmo tempo. Para poder tratar a conexão de vários clientes simultaneamente, Java usa threads.

Veja como ficaria o servidor anterior, agora recebendo a conexão de vários clientes simultaneamente (o código abaixo refere-se apenas ao servidor. Continue usando o cliente da seção 4.1.2):

```
import java.net.*;
import java.io.*;
public class MultiServer {
   public static void main (String args []) {
      Socket client;
      ServerSocket server;
      try {
             server = new ServerSocket (5000);
             while (true) {
                System.out.println ("Esperando conexão de cliente!");
                client = server.accept();
                new ClienteThread (client).start();
          }
      }
      catch (IOException ioe) { }
  }
}
/* ********************************************** */
class ClienteThread extends Thread {
   Socket client;
   public ClienteThread (Socket sock) {
             client = sock;
   }

   public void run () {
      try {
          System.out.println ("Cliente se conectando.");
          InputStream in = client.getInputStream (); // obtendo stream de entrada
          BufferedReader entrada = new BufferedReader (new InputStreamReader (in));
          String str=null;
          while (true) {
              str = entrada.readLine();
              if (str==null || str.equals("BYE"))  {
                  System.out.println ("Conexao finalizada.");
                  break;
              }
              else System.out.println ("Recebeu: "+str);
          }
          if (str!=null) client.close ();
       }
        catch (IOException ioe) { }
   }
} // class
```

1. Os applets só podem estabelecer conexões de socket com o computador de onde foram descarregados.
2. Para obter o endereço IP de sua máquina, você pode digitar winipcfg no menu Iniciar → Executar.
3. Se o programa servidor está funcionando na mesma máquina que o programa cliente, você pode especificar o endereço IP como "127.0.0.1" ou "localhost". Com isso, você está dizendo que é o endereço IP da máquina local. Por exemplo:
Socket s = new Socket ("localhost", 5000);

4.2 Fazendo conexões com URL

Neste capítulo discutiremos como o seu programa Java pode usar URLs para acessar informação na Internet. Uma URL (Uniform Resource Locator) é um endereço de um recurso na Internet. O programa Java pode usar URL para ler e enviar informações para este recurso, por exemplo, ler o conteúdo de uma página HTML em um determinado site. Neste capítulo veremos uma definição mais detalhada de URL, como criar uma URL e os seus métodos e como criar uma conexão a uma URL para ler e enviar dados (http://java.sun.com/docs/books/tutorial/networking/urls/index.html).

4.2.1 O que é uma URL?

Geralmente, nós pensamos em uma URL como o nome de um arquivo na World Wide Web. Isso porque muitas vezes ela se refere a algum arquivo em alguma máquina na rede. Mas, na realidade, uma URL pode acessar outros recursos, como uma conexão a rede, um serviço gopher e etc.

URL é uma abreviação para *Uniform Resource Locator* e é uma referência para algum recurso na Internet.

Uma URL é formada por dois componentes: o idenficador do protocolo e o nome do recurso. Por exemplo:

O identificador de protocolo indica o nome do protocolo que deve ser usado para obter o recurso. O exemplo acima usa Hypertext Transfer Protocol (HTTP), o qual é utilizado para obter documentos hipertextos. Além de HTTP existem ainda File Transfer Protocol (FTP), Gopher, File, e News.

O nome do recurso é composto pelo endereço completo do recurso. O formato do nome vai depender do protocolo usado. Para a maioria dos protocolos, inclusive HTTP, o nome vai ser formado por um ou mais dos seguintes componentes:

Nome da Máquina: O nome da máquina onde o recurso está localizado.
Nome do arquivo: O path do arquivo na máquina.
Número da porta: A porta a se conectar (opcional).
Referência:Uma referência para uma âncora dentro do arquivo que especifica uma determinada localização dentro do arquivo (opcional).

4.2.2 Criando um objeto URL

Dentro dos programas Java, podemos criar um objeto que representa um endereço de URL. Podemos criar um objeto URL fornecendo uma String com o endereço URL. Por exemplo, a URL para o site da Gamelan é:

http://www.gamelan.com/

O objeto URL seria criado da seguinte maneira:

```
URL gamelan = new URL("http://www.gamelan.com/");
```

O objeto URL criado acima representa uma URL absoluta que contém todas as informações necessárias para acessar o recurso. Podemos ainda criar um objeto URL a partir de um endereço de URL relativo.

4.2.3 Criando uma URL relativa a outra

Uma URL relativa contém apenas parte da informação necessária para acessar um recurso. Veja os exemplos de utilização de URL relativas para os endereços abaixos:

http://www.gamelan.com/pages/Gamelan.game.html

http://www.gamelan.com/pages/Gamelan.net.html

Você pode criar objetos URL para essas páginas relativas ao endereço da máquina (host). Isso poderia ser feito da seguinte maneira:

```
URL gamelan = new URL("http://www.gamelan.com/pages/");
URL gamelanGames = new URL(gamelan, "Gamelan.game.html");
URL gamelanNetwork = new URL(gamelan, "Gamelan.net.html");
```

Os exemplos abaixo utilizam um construtor que permite criar um objeto URL a partir de outro:

```
URL(URL baseURL, String relativeURL)
```

4.2.4 Tratando exceções em URL

Ao criar um objeto URL devemos tratar a exceção MalformedURLException, no caso de o endereço se referir a um protocolo desconhecido. Por exemplo:

```
try {
    URL myURL = new URL(. . .)
} catch (MalformedURLException e) {
    // código de tratamento de exceção aqui
}
```

URL são objetos "write-once", uma vez criados, os valores de seus atributos não podem ser modificados (protocolo, nome do *host*, nome do arquivo ou número da porta).

4.2.5 Métodos de URL

O objeto URL contém vários métodos que podem ser usados para obter os componentes desta URL.

Métodos	Descrição
String getProtocol ( )	retorna o identificador de protocolo da URL.
String getHost ( )	retorna o nome da host.
int getPort ( )	retorna um inteiro que representa o número da porta. Senão foi especificado o número de porta retorna –1.
String getFile ( )	retorna nome do arquivo.
String getRef ( )	retorna nome da referência.

Por exemplo:

```
import java.net.*;
import java.io.*;

public class ParseURL {
public static void main(String[] args) throws Exception {
        URL aURL = new
URL("http://java.sun.com:80/docs/books/tutorial/intro.html#DOWNLOADING");
        System.out.println("protocol = " + aURL.getProtocol());
        System.out.println("host = " + aURL.getHost());
        System.out.println("filename = " + aURL.getFile());
        System.out.println("port = " + aURL.getPort());
        System.out.println("ref = " + aURL.getRef());
    }
}
```

O resultado é:


```
protocol = http
host = java.sun.com
filename = /docs/books/tutorial/intro.html
port = 80
ref = DOWNLOADING
```

4.2.6 Lendo diretamente de uma URL

Após criar uma URL, podemos chamar o método `openStream()` da URL para obter uma stream de onde se possa ler o conteúdo da URL. O método `openStream()` retorna um objeto `java.io.InputStream`.

O programa seguinte usa o método `openStream()` para obter uma sream de entrada da URL http://www.yahoo.com/.

```
import java.net.*;
import java.io.*;
public class URLReader {
    public static void main(String[] args) throws Exception {
       URL yahoo = new URL("http://www.yahoo.com/");
       BufferedReader in = new BufferedReader(
                           new InputStreamReader(
                           yahoo.openStream()));
       String inputLine;
       while ((inputLine = in.readLine()) != null)
           System.out.println(inputLine);
       in.close();
    }
}
```

4.2.7 Conexão a uma URL

Após criar um objeto URL, temos que chamar o método `openConnection( )` para se conectar a URL. Quando nos conectamos a uma URL estamos iniciando um link de comunicação entre o programa Java e a URL através da Internet. Por exemplo, vamos abrir uma conexão ao site do Yahoo:

```
try {
    URL yahoo = new URL("http://www.yahoo.com/");
    yahoo.openConnection();
} catch (MalformedURLException e) {     // new URL() failed
    . . .
} catch (IOException e) {               // openConnection() failed
    . . .
}
```

Se possível, o método `openConnection( )` cria uma conexão com a URL e retorna um novo objeto `URLConnection` (se já não existe um). Se alguma coisa der errada, por exemplo, o servidor Yahoo estiver fora do ar, o método `openConnection( )` gera uma exceção IOException.

Após ter se conectado a URL, podemos usar o objeto `URLConnection` para realizar ações como ler da e escrever para a conexão, assunto da próxima seção.

4.2.8 Reading from and Writing to a URLConnection

A classe `URLConnection` contém muitos métodos que podem ser usados para a comunicação com a URL através da rede Internet. `URLConnection` é uma classe voltada para conexões HTTP, por isso, muitos dos métodos disponibilizados nesta classe só podem ser usados para trabalhar com URLs HTTP. A maioria dos protocolos permite que se possa ler e escrever na conexão.

Reading from a URLConnection

O programa que mostraremos como exemplo tem a mesma função que o programa URLReader, visto anteriormente. Entretanto, ao invés de ler de uma stream de entrada diretamente da URL, este programa explicitamente abre uma conexão e pega uma stream de entrada da conexão. Os comandos em negrito realçam a diferença entre este programa e o URLReader.

```
import java.net.*;
import java.io.*;

public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                yc.getInputStream()));
        String inputLine;

        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

A saída deste programa é idêntica a saída do programa que abre uma stream diretamente da URL. Você pode usar qualquer um dos dois para ler de uma URL. Entretanto, ler de uma `URLConnection` ao invés de ler diretamente de uma URL pode ser mais útil. Isto porque se pode utilizar o objeto `URLConnection` para outras tarefas (como escrever em uma URL) ao mesmo tempo.

Escrevendo em uma URLConnection

Muitas páginas HTML contêm formulários (*forms)* – campos de texto e outros objetos de interface - que permitem a entrada de dados para envio de informações ao servidor. Após você digitar as informações necessárias e submetê-las (geralmente clicando no botão submit), o browser envia os dados pela rede que são recebidos em um script `cgi-bin` no servidor Web. O programa cgi-bin recebe os dados, os processa e envia uma resposta, geralmente, uma nova página HTML.

Um programa Java também pode enviar dados para um script `cgi-bin no servidor`. Para tanto ele deve:

Criar uma URL.
Abrir uma conexão a uma URL.
Determinar as funcionalidades de saída no URLConnection.
Obter uma stream de saída da conexão.
Escrever na stream de saída.
Fechar a stream de saída.

Para testar o envio de informações através de uma conexão URL, nós vamos utilizar um cgi chamado `backwards` que se encontra na página http://java.sun.com/cgi-bin/backwards. Este script lê uma string de sua entrada padrão, inverte as ordens dos caracteres da string e escreve o resultado na sua saída padrão.

O programa a seguir envia a string através da URL para o script backwards através de um objeto `URLConnection`:

```
import java.io.*;
import java.net.*;

public class Reverse {
    public static void main(String[] args) throws Exception {

       if (args.length != 1) {
           System.err.println("Usage:  java Reverse string_to_reverse");
           System.exit(1);
       }

       String stringToReverse = URLEncoder.encode(args[0]);

       URL url = new URL("http://java.sun.com/cgi-bin/backwards");
       URLConnection connection = url.openConnection();
       connection.setDoOutput(true);

       PrintWriter out = new PrintWriter(connection.getOutputStream());
       out.println("string=" + stringToReverse);
       out.close();

       BufferedReader in = new BufferedReader(
                           new InputStreamReader(
                           connection.getInputStream()));
       String inputLine;
```

```
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);

        in.close();
    }
}
```

Vamos entender o programa. Primeiro, ele verifica se o usuário passou algum argumento de linha de comando, ou seja, a string que vai ser passada para o script.

```
if (args.length != 1) {
    System.err.println("Usage:  java Reverse " +
                       "string_to_reverse");
    System.exit(-1);
}
String stringToReverse = URLEncoder.encode(args[0]);
```

Esta string tem que ser codificada para ser processada no servidor. Os métodos da classe `URLEncoder` codificam os caracteres da string num formato apropriado para serem recebidos no servidor.

Após, o programa cria o objeto `URL` para o script `backwards`, abre uma conexão e seta a conexão para escrita.

```
URL url = new URL("http://java.sun.com/cgi-bin/backwards");
URLConnection c = url.openConnection();
c.setDoOutput(true);
```

O programa cria uma stream de saída na conexão e abre um `PrintWriter` nele:

```
PrintWriter out = new PrintWriter(c.getOutputStream());
```

Se a URL não suportar escrita (saída), o método `getOutputStream` gera uma exceção `UnknownServiceException`. Se a URL suporta saída, então o método retorna uma stream de saída que está conectada na stream de entrada do servidor, ou seja, a saída do cliente está ligada a entrada do servidor.

Após, o programa escreve a informação para a saída de stream e fecha a conexão:

```
out.println("string=" + stringToReverse);
out.close();
```

O script no servidor irá ler a informação do cliente, realizar algum processamento e retornar uma informação ao cliente. Então, é necessário que após o programa Java cliente enviar informações ao servidor, ele possa ler da URL o resultado enviado pelo servidor.

```
BufferReader in = new BufferedReader(new InputStreamReader(c.getInputStream()));
String inputLine;
while ((inputLine = in.readLine()) != null)
    System.out.println(inputLine);
in.close();
```

Abaixo é exibido um exemplo de saída desse programa.


```
c:\java Reverse "Reverse Me"
Reverse Me
 reversed is:
eM esreveR
```

5 Remote Method Invocation (RMI)

Através da *Remote Method Invocation (RMI)* é possível chamar métodos em um servidor remoto de maneira transparente, como se o objeto estivesse na máquina local.

Um objeto remoto executa em um servidor. Cada objeto remoto implementa uma interface remota que especifica quais dos seus métodos podem ser invocados por clientes (objetos em outras máquinas). Os clientes podem invocar os métodos de objetos remotos como se estivessem chamando métodos locais. Por exemplo, um cliente pode pedir para um objeto banco de dados somar os valores de um conjunto de registros e retornar os resultados. Isto é mais eficiente que copiar localmente todos os registros do banco de dados e depois somar.

Na perspectiva do programador, objetos e métodos remotos funcionam tal como os métodos e objetos locais. Os detalhes de implementação ficam escondidos. As etapas necessárias para chamar um objeto remoto são:

importar o pacote java.rmi.*;
consultar o objeto remoto em um registro (registry);
tratar a exceção RemoteException quando o método do objeto remoto é chamado.

Os objetos remotos podem implementar uma ou mais interfaces remotas que vão definir os métodos que podem ser chamados remotamente.

5.1 Stubs e Skeletons

O cliente pode invocar o método remoto usando o stub. O stub é um objeto especial que implementa as interfaces remotas do objeto remoto. Desta maneira, o stub possui métodos com a mesma assinatura que os métodos do objeto remoto. O cliente pensa que está chamando um método remoto, mas na verdade está chamando um método do stub. Stubs são usados na máquina cliente no lugar do objeto remoto que vive no servidor. Quando o stub é invocado, ele invoca o método do objeto remoto no servidor.

O stub envia a informação para o servidor. Para tanto, ele encapsula os parâmetros usados no método remoto em um bloco de bytes. Esse processo de codificar os dados enviados em um formato apropriado para transmissão é chamado de *marshaling*.

No servidor, a requisição é recebida pelo skeleton. O skeleton lê os argumentos e faz a chamada do método no servidor. Se a chamada do método retorna um valor, o valor é passado para o skeleton e recebido pelo stub na máquina cliente.

Assim, como podemos limitar as ações que podem ser feitas por um applet na máquina do usuário, é possível definir uma política de segurança para objetos remotos no servidor. Para isso, um objeto SecurityManager checa se todas as operações do cliente são permitidas pelo servidor. Além disso, é possível exigir identificação e determinar diferentes níveis de acesso.

5.2 O Primeiro Exemplo

Para entender melhor, vejamos o exemplo retirado de (Harold, 1997). Neste exemplo, um servidor (Server.java) disponibiliza um objeto remoto com o nome "HelloServer" (classe

ObjetoRemoto.java). O programa cliente (Cliente.java) chama o método sayHello() do objeto remoto que retorna a String "Hello World!".

5.2.1 Definindo a Interface

O programa cliente precisa manipular um objeto que está no servidor (ObjetoRemoto.java). Mas, para manipular este objeto ele precisa saber o comportamento deste objeto. Este comportamento está definido na interface que contém a declaração de todos os métodos do objeto remoto que podem ser acessados pelo programa cliente.

Todas as interfaces de objetos remotos devem ser derivadas da interface Remote. Além disso, todos os métodos desta interface devem tratar a exceção RemoteException. Isto é necessário caso a conexão do servidor esteja indisponível. Assim, o cliente tem de estar preparado para situações como esta.

```
public interface InterfaceRemota extends java.rmi.Remote {
       String sayHello() throws java.rmi.RemoteException;
}
```

5.2.2 Definindo o código do Objeto Remoto

Agora, no servidor, deve ser implementada a classe do objeto remoto que será acessado pelo programa cliente. O objeto deve ser derivado da classe UnicastRemoteObject, o que torna o objeto disponível remotamente. O único método disponível na classe é o sayHello().

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class ObjetoRemoto extends UnicastRemoteObject implements InterfaceRemota {
  private String name;
  public ObjetoRemoto (String s) throws java.rmi.RemoteException {
    super();
    name = s;
  }
  public String sayHello() throws RemoteException {
    return  "Hello World!";
  }
}
```

5.2.3 Criando o servidor

Para que um programa cliente acesse um objeto no servidor é preciso possuir um mecanismo que lhe permita saber os objetos remotos que estão disponíveis no servidor. Para isso ele pode usar o registry. O registry mantém uma lista dos objetos disponíveis no servidor RMI e o nome pelo qual eles podem ser acessados. No programa servidor abaixo, usamos o método rebind( ) para adicionar um objeto do tipo ObjetoRemoto com o nome de HelloServer no registry Naming.

```
import java.rmi.*;
class Servidor {
  public static void main(String args[]) {
    try {
      ObjetoRemoto obj = new ObjetoRemoto("HelloServer");
      Naming.rebind("HelloServer", obj);
      System.out.println("ObjetoRemoto foi criado e registrado");
    }
    catch(Exception e) {
      System.out.println("Ocorreu uma excecao no servidor:");
    }
  }
}
```

5.2.4 Executando um programa cliente para acessar um objeto remoto

Antes do programa cliente chamar o objeto remoto, ele precisa recuperar uma referência do objeto remoto. Essa referência é recuperada chamando o método lookup (String nome) da classe Naming. O método lookup ( ) da classe Naming retorna uma referência remota que permite ao cliente invocar métodos do objeto remoto. Quando este método é chamado ele acha o stub do objeto solicitado no servidor pelo rmiregistry (que estará funcionando na porta 1099) e o copia localmente. Neste método, fornecemos a URL do servidor e o nome em que o objeto foi cadastrado no servidor, no caso, HelloServer.

```
InterfaceRemota obj = (InterfaceRemota) Naming.lookup("//" + "localhost" +
"/HelloServer");
```

O objeto que é recuperado perde o seu tipo de informação, por isso, é preciso usar um casting ao receber o objeto. Uma vez que o objeto remoto foi recuperado, é possível chamar o método desejado.

```
String msg = obj.sayHello();
```

```
import java.awt.*;
import java.rmi.*;
public class Cliente {
  public static void main (String args[]) {
    try {
       InterfaceRemota obj = (InterfaceRemota) Naming.lookup("//" +
                                "localhost" + "/HelloServer");
       String message = obj.sayHello();
       System.out.println ("Recebeu do servidor: "+message);
    }
    catch(Exception e) {
      System.out.println("Ocorreu uma excecao:");
    }
  }
}
```

Para executar o programa acima, podemos usar um arquivo ".bat" que deverá realizar as seguintes ações:

compilar todos os fontes java: javac *.java
ativa sistema de registro: start /min rmiregistry
cria stubs e skeletons: rmic ObjetoRemoto
inicia programa servidor: start java Servidor
aguarda servidor acabar cadastro do objeto remoto: pause
executa programa cliente: java Cliente

Caso o servidor e o cliente sejam executados em máquinas diferentes, a classe Servidor, a interface remota, o objeto remoto e as classes do stub e do skeleton devem ser copiadas para o servidor. Na máquina cliente devem existir o programa cliente, a interface e a classe do stub.

O programa cliente acima gera a seguinte saída:


```
C:\ >java Cliente

Recebeu do servidor: Hello World!
```

5.3 Segundo Exemplo – Modificando Valores do Objeto Remoto no Servidor

A implementação a seguir, baseada em (Horstman & Cornell, 2000b), mostra um exemplo de um programa cliente que modifica os valores de atributo de um determinado objeto remoto. Primeiramente, vamos definir a interface do objeto remoto e sua implementação.

```
import java.rmi.*;
public interface Produto extends Remote {
  // Informa a descrição do produto
  String leDescricao() throws RemoteException;
  // Informa o valor do produto
  float leValor() throws RemoteException;
  // Altera a descrição do produto
  void setaDescricao(String s) throws RemoteException;
  // Altera o valor do produto
  void setaValor(float v) throws RemoteException;
}
```

```
import java.rmi.*;
import java.rmi.server.*;
public class ProdutoImplementacao extends UnicastRemoteObject implements Produto {
  private String nome;
  private float valor;
  public ProdutoImplementacao(String nome, float valor) throws RemoteException {
    this.nome = nome;
    this.valor = valor;
  }
  public String leDescricao() throws RemoteException {
    if(nome.equals("televisao"))
      return nome + " Panasonic";
    else if(nome.equals("geladeira")) return nome + " Brastemp";
    else   return nome + " e' um produto novo. Excelente !";
  }
  public float leValor() throws RemoteException {
    return valor;
  }
  public void setaValor(float valor) throws RemoteException {
    this.valor = valor;
  }
  public void setaDescricao(String nome) throws RemoteException {
    this.nome = nome;
  }
}
```

O programa servidor cadastra dois objetos remotos do tipo ProdutoImplementação.

```
public class ProdutoServidor {
  public static void main(String args[]) {
    try {
      ProdutoImplementacao p1 = new ProdutoImplementacao("geladeira", 1200.00F);
      ProdutoImplementacao p2 = new ProdutoImplementacao("televisao", 400.00F);
      // OBS.: O nome registrado dos produtos deve ser exclusivo.
      java.rmi.Naming.rebind("PrimeiroProduto", p1);
      java.rmi.Naming.rebind("SegundoProduto", p2);
      System.out.println ("Produtos registrados.");
    }
    catch(Exception e) {
      System.out.println("Erro: " + e);
      e.printStackTrace();
    }
  }
```

```
}
```

O programa cliente irá obter a referências destes objetos e modificar os atributos de uma das referências.

```
/*Criando um Cliente para pegar informações sobre diversos produtos de um Servidor
remoto.*/
import java.rmi.*;
public class ProdutoCliente {
  public static void main(String[] args) {
    String url = "localhost/";
    try {
      Produto p1 = (Produto)Naming.lookup("rmi://" + url + "PrimeiroProduto");
      Produto p2 = (Produto)Naming.lookup("rmi://" + url + "SegundoProduto");
      System.out.println(p1.leDescricao());
      System.out.println(p2.leDescricao());
      System.out.println("R$ = " +  p1.leValor());
      System.out.println("R$ = " +  p2.leValor());
      p2.setaValor((float)54.56);
      p2.setaDescricao("maquina de lavar");
      System.out.println("R$ = " +  p2.leValor());
      System.out.println(p2.leDescricao());
    }
    catch(Exception e) {
      System.out.println("Erro: " + e);
      e.printStackTrace();
    }
  }
}
```

A seguir podemos executar o programa através dos seguintes comandos:

```
javac *.java
start /min rmiregistry
rmic ProdutoImplementacao
start Java ProdutoServidor
pause
java ProdutoCliente
```

O programa cliente acima produz o seguinte resultado:


```
C:\ >java ProdutoCliente
geladeira Brastemp
televisao Panasonic
R$ = 1200.0
R$ = 400.0
R$ = 54.56
maquina de lavar e' um produto novo. Excelente !
```

O SegundoProduto teve a sua descrição modificada para "maquina de valor" e o seu valor para 54.56. Estes valores foram alterados no objeto cadastrado no servidor. Se fosse executado um outro programa cliente, pedindo os valores dos atributos do objeto SegundoProduto, seriam fornecidos os novos valores.

5.4 Transmitindo Objetos entre Cliente e Servidor

Quando um objeto é passado ou retornado de um método Java, o que é realmente transferido é a referência ao objeto. Em Java, referências são apontadores para a localização de um objeto na memória da máquina virtual Java. Mas, uma máquina remota não pode acessar a memória de uma máquina local. Então, como passar objetos?

Existem duas maneiras. Pode ser passado uma referência remota ao objeto (uma referência que aponta para a memória da máquina remota) ou passar uma cópia do objeto.

Para copiar um objeto, é necessário converte-lo em uma stream de bytes para ser enviado para a máquina remota. Isto pode ser complicado, pois um objeto pode conter outros objetos como campos, que devem ser passados. Serialização de objetos é um esquema que converte objetos em um stream de bytes que é enviado para outras máquinas. No receptor, este objeto é reconstruído a partir dos bytes recebidos. Os objetos serializados também podem ser gravados em discos e recuperados posteriormente. Todos os tipos primitivos Java e objetos remotos podem ser serializados. Além desses, os tipos Java que implementam Serializable são String, AWT componentes, Date, JDBC ResultSet e outros[1].

Assim, como em programas locais, também é necessário passar parâmetros (que podem ser objetos) para os objetos remotos. Isto pode ser feito de três maneiras:

Tipos primitivos (char, int, etc) são passados por valor (by value), assim como na chamada de métodos locais.

Referência a objetos remotos (objetos que implementam a interface Remote) é passada como uma referência remota que permite que o receptor invoque métodos do objeto remoto. Isso é similar como é passada a referência de objetos na chamada de métodos locais, com a diferença que a referência aponta um objeto na memória da máquina remota.

Objetos que não implementam a interface Remote são passados por valor (by value), ou seja, é feita uma cópia do objeto (por serialização) e enviada para o cliente.

[1] *Para uma referência completa de tipos Java que implementam Serializable, ver livro Java Network Programming de Elliot Harold, na página 350.*

6 JDBC – Conectividade com Banco de Dados em Java

JDBC significa Java Database Connectivity (Conectividade em Banco de Dados em Java) e é uma API que permite a um programa Java acessar um banco de dados, ou seja, uma interface entre a linguagem Java e a linguagem que os outros banco de dados suportam.

A maior vantagem de JDBC sobre os outros ambientes de programação de banco de dados é que os programas desenvolvidos em Java com JDBC são independentes de plataforma ou do banco de dados. Isso significa que um programa Java que acessa uma base de dados pode funcionar tanto em Windows como Unix, como também pode acessar qualquer banco de dados, como Oracle, o Microsoft SQL Server e outros.

Mas como resolver o problema de Java acessar diferentes bancos de dados com diferentes protocolos de comunicação. É impossível fazer com que Java acesse os bancos de dados diretamente com "java puro", pois existem muitos bancos de dados e, por isso, muitos protocolos que deveriam ser implementados.

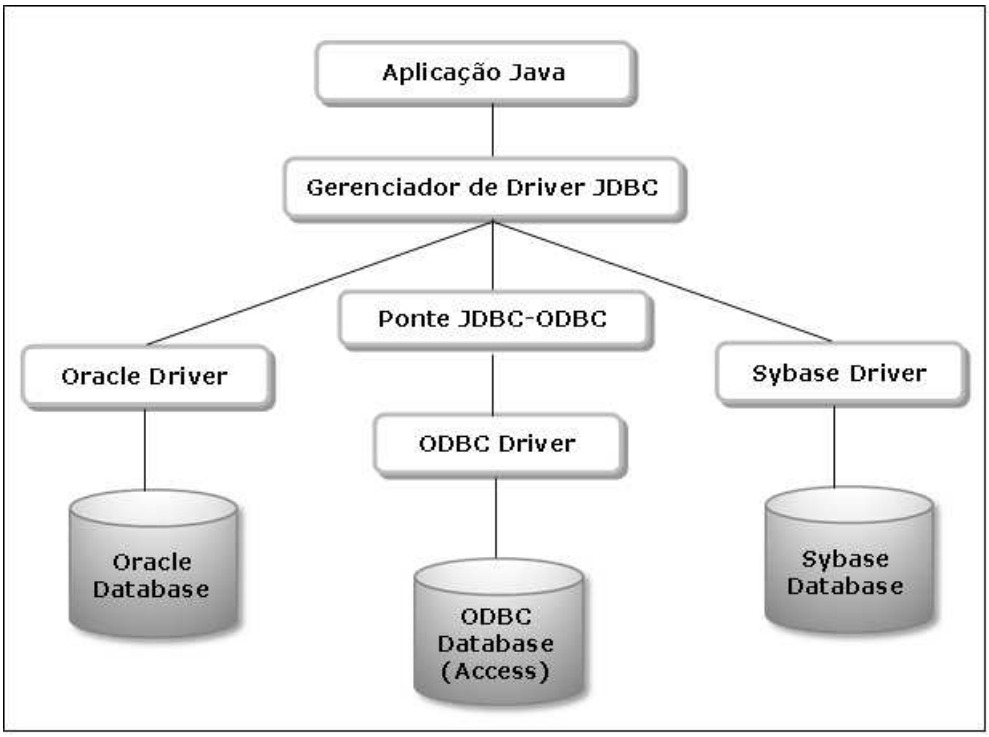

Figura 3: Java e Banco de Dados

A maneira que Java encontrou de resolver isso é fornecendo uma API puramente Java para acesso SQL, juntamente com um gerenciador de driver que se conecta a um driver proprietário do banco de dados. Tudo o que os desenvolvedores de banco de dados tem de fazer é fornecer um driver específico do seu banco de dados que se comunique com o gerenciador de drivers do Java, conforme ilustra Figura 3.

Por exemplo, digamos que tenhamos um programa Java que consulta uma base de dados Oracle. É necessário temos o driver JDBC da Oracle instalado na nossa máquina[2]. Quando for realizada uma consulta, o programa Java envia as declarações SQL para o gerenciador de driver JDBC que envia a consulta para o driver JDBC do Oracle que, então, se conectará a base de dados Oracle para retornar o resultado da consulta ou realizar a ação solicitada.

6.1 Trabalhando com JDBC em plataforma Windows

O pacote do JDBC vem juntamente com o JDK. Se você já instalou o JDK em sua máquina, você já tem o JDBC instalado. Uma segunda etapa é instalar o driver JDBC da base de dados a ser usada. Como vamos criar um aplicativo que acessa uma base de dados Access, nós vamos usar o ODBC driver. O ODBC driver, assim como o JDBC, é um driver que se comunica com vários bancos de dados. Para Java se comunicar com o driver ODBC existe a ponte JDBC-ODBC, que já vem com o pacote JDBC no JDK.

É necessário tornar a base de dados experimental em uma fonte de dados ODBC. Os seguintes passos devem ser realizados no Windows:

Acessar o Painel de Controle e ativar o aplicativo ODBC (Menu Iniciar→ Configurações → Painel de Controle → Fonte de Dados ODBC);

Clique na orelha DNS do usuário e botão Adicionar;

Irá aparecer uma lista de drivers. Você deve escolher o driver a qual deseja adicionar um novo banco de dados, no caso, Microsoft Access. Após isso, clique no botão Concluir.

---

[2] *Uma lista de drivers JDBC disponíveis pode ser encontrado no site: http://splash.javasoft.com/jdbc/jdbc.drivers.html*

Na nova caixa de diálogo você deve fornecer o nome pelo qual a base de dados vai ser acessada no programa Java na caixa Nome da Fonte de dados. Se você já criou a base de dados, clique no botão Selecionar e especifique a localização do arquivo. Se você deseja criar uma base de dados, clique no botão Criar e forneça o nome e a localização desejada para o arquivo.

Para entender melhor, vejamos o exemplo de um programa que cria uma tabela com o nome Empregados. Nesta tabela são armazenados os dados dos empregados que são um identificador inteiro, nome e salário.

```
import java.sql.*;
public class JdbcAccess {
  public static void main(String args []) throws SQLException {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
```

```
    catch(ClassNotFoundException e) {
      System.out.println("Não Consigo carregar o driver!!!");
      e.printStackTrace();
      return;
    }
    Connection conn = DriverManager.getConnection("jdbc:odbc:CursoJava", "", "");
   Statement stmt = conn.createStatement();
   String comando = "CREATE TABLE Empregados (nome CHAR(35), id INT, salario
DOUBLE)";
   stmt.executeUpdate(comando);
   stmt.close();
   conn.close();
  }
}
```

6.2 Carregando o driver JDBC

Em uma primeira etapa é preciso carregar o driver JDBC do banco de dados. Para a ponte JDBC-ODBC, o nome do driver é "sun.jdbc.odbc.JdbcOdbcDriver". Para carregar o driver use o comando:

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
```

Se estiver usando um outro banco de dados, então procure o nome do driver e o carregue ao invés deste.

6.3 Fazendo a conexão a um banco de dados

Após carregar o driver, deve ser feita a conexão com o banco de dados. Para isso, deve ser fornecida a URL do banco de dados. Essa URL tem o seguinte formato:

```
jdbc:nome_subprotocolo://url:porta
```

Onde nome_subprotocolo depende do driver JDBC do banco de dados que está sendo usado.

Para fazer a conexão, usa-se o comando:

```
Connection conn = DriverManager.getConnection("jdbc:odbc:CursoJava", user,
password);
```

user e password são strings e são usadas apenas quando é necessário realizar uma identificação para acessar o banco de dados.

O método getConnection() retorna um objeto Connection que permite realizar consultas.

6.4 Fazendo consultas

Para realizar consultas é necessário criar um objeto *Statement*. Este objeto é obtido pela chamada do método createStatement() do objeto Connection.

```
Statement stmt = conn.createStatement();
```

Este objeto Statement recebido pode ser usado para múltiplas consultas.

O objeto statement possui dois métodos principais usados na consulta a banco de dados:

ResulSet executeQuery (String): usado para fazer consultas SELECT. Retorna um objeto ResultSet que contém o resultado da consulta.

int executeUpdate (String): usado para fazer INSERT, UPDATE, DELETE e CREATE TABLE. Retorna um inteiro com o número de registros afetados.

Um exemplo de consulta é:

```
String query = "SELECT First_Name, Last_Name" +
"FROM Employees" +
"WHERE Last_Name LIKE 'Washington' ";
ResultSet rs = stmt.executeQuery (query);
```

Você deve ter notado que as palavras-chave da consulta SQL estão em letras maiúsculas. SQL não requer isso, é apenas uma convenção usada para deixar o código mais claro. Por exemplo, os dois comandos SQL abaixo executam da mesma maneira sem retornar erro.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington'
select First_Name, Last_Name
from Employees
where Last_Name like 'Washington '
```

Mas, os valores para campos são case-sensitive. Nesse caso o nome "Washington" deve ser escrito com "W" maiúsculo e as outras letras em minúsculo.

Os requisitos para a identificação de nomes de tabelas e campos variam de banco de dados para banco de dados. Por exemplo, alguns bancos de dados requerem que os nomes de tabelas e campos sejam dados exatamente como criados na declaração CREATE TABLE, outros não. Por isso, o melhor é sempre fornecer os nomes na forma em que foram criados, respeitando-se maiúsculas e minúsculas.

6.5 Obtendo os Resultados das Consultas

O resultado da consulta é retornado em um objeto do tipo ResultSet. Uma consulta pode ter vários registros com vários campos. Para acessar o próximo registro (linha) da consulta, chame o método next( ) do ResultSet. Note que JDBC não fornece um método previous( ), por isso não é possível mover para o registro anterior em um objeto do tipo ResultSet. Para acessar cada campo, use o método getXXXX( ).

```
ResultSet rs =stmt.executeQuery ("SELECT * FROM Empregados");
While (rs.next ()) {
       String nome = rs.getString (1);
          double salario = rs.getDouble ("salario");
}
```

Os métodos getXXX( ) permitem acessar os diferentes tipos de dados provenientes do resultado da consulta. Abaixo, você pode ver uma tabela dos tipos de dados JDBC (SQL) correspondentes aos tipos Java e os métodos getXXX( ) usados para obtê-los:

Tipo de dado JDBC	Tipo de dado Java	Método getXXX
INTEGER ou INT	int	getInt()
SMALLINT	short	getShort()
REAL	float	getFloat()
DOUBLE	double	getDouble ()
CHARACTER(n), CHAR(n) ou VARCHAR(n)	String	getString()
DATE	java.sql.Date	getDate()
TIME	java.sql.Time	getTime()
BIN	boolean	getBoolean()

Podem haver pequenas variações entre os tipos SQL de diferente banco de dados. Por exemplo, os maiores banco de dados suportam dados binários grandes, mas Orcale chama de LONG RAW, Sybase de IMAGE, Informix de BYTE e DB2 de LONG VARCHSR FOR BIT DATA.

Felizmente, o programador não precisará preocupar-se com o tipo SQL usado pelo seu banco de dados. A maioria dos programadores já estará trabalhando com banco de dados existentes, então eles não precisam se preocupar com o nome SQL usado para criar estas tabelas.

JDBC define um conjunto de tipos SQL genéricos identificados na classe java.sql.Types. Estes tipos foram projetados para representar os tipos SQL mais usados. Os programas JDBC poderão usar esses tipos para referenciar tipos SQL genéricos, sem se preocupar com o nome exato do tipo SQL usado pelo banco de dados com que está se trabalhando.

Os programadores devem ter maior preocupação em usar os nomes de tipos SQL na declaração CREATE TABLE quando estão criando uma tabela de banco de dados. Neste caso, os programadores devem tomar o cuidado para utilizar os nomes de tipo SQL suportados pelo banco de dados utilizado. Para isso, é melhor consultar a documentação do banco de dados para saber os seus tipos SQL. Na Tabela 1 podemos observar os tipos JDBC mapeados em tipos SQL específicos dos bancos de dados mais conhecidos.

Se você desejar criar programas JDBC portáveis que podem criar tabelas em diferentes bancos de dados, você tem duas escolhas. Primeiro você se restringe a usar apenas os tipos SQL mais aceitáveis tais como INTEGER, NUMERIC ou VARCHAR, que tem funcionamento semelhante para todos os bancos de dados. Em segundo, você pode usar o método java.sql.DataBaseMetaData.getTypeInfo( ) para descobrir quais tipos SQL são suportados por um banco de dados e selecionar um nome de tipo SQL específico do banco de dados que corresponda ao tipo JDBC.

JDBC define um padrão para mapear tipos JDBC para tipos Java. Por exemplo, o tipo JDBC INTEGER é normalmente mapeado para o int Java. Isto fornece uma interface para ler e armazenar valores como tipos Java.

JDBC Type Name	Oracle 7.2	Sybase 11.0	Informix 7.12	IBM DB2 2.1
BIT		BIT		
TINYINT		TINYINT		
SMALLINT	SMALLINT	SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER	INTEGER	INTEGER
BIGINT				
REAL	REAL	REAL	REAL	
FLOAT	FLOAT	FLOAT	FLOAT	FLOAT
DOUBLE	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION
NUMERIC(p,s)	NUMERIC(p,s)	NUMERIC(p,s)	NUMERIC(p,s)	NUMERIC(p,s)
DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)	DECIMAL(p,s)
CHAR(n)	CHAR(n) n <= 255	CHAR(n) n <= 255	CHAR(n) n <= 32,767	CHAR(n) n <= 254
VARCHAR(n)	VARCHAR(n) n <= 2000	VARCHAR(n) n <= 255	VARCHAR(n) n <= 255	VARCHAR(n) n <= 4000
LONGVARCHAR	LONG VARCHAR limit is 2 Gigabytes	TEXT limit is 2 Gigabytes	TEXT limit is 2 Gigabytes	LONG VARCHAR limit is 32,700 bytes
BINARY(n)		BINARY(n) n <= 255		CHAR(n) FOR BIT DATA n <=254
VARBINARY	RAW(n) n <= 255	VARBINARY(n) n <= 255		VARCHAR(n) FOR BIT DATA n <= 4000
LONGVARBINARY	LONG RAW limit is 2 Gigabytes	IMAGE limit is 2 Gigabytes	BYTE limit is 2 Gigabytes	LONG VARCHAR FOR BIT DATA limit is 32,700 bytes
DATE	DATE	DATETIME	DATE	DATE
TIME	DATE	DATETIME		TIME
TIMESTAMP				TIMESTAMP

JDBC Type Name	Microsoft SQL Server 6.5	Microsoft Access 7.0	Sybase SQL Anywhere 5.5
BIT	BIT	BIT	BIT
TINYINT	TINYINT	BYTE	TINYINT
SMALLINT	SMALLINT	SMALLINT	SMALLINT
INTEGER	INTEGER	INTEGER	INTEGER
BIGINT			
REAL	REAL	REAL	REAL
FLOAT	FLOAT	FLOAT	FLOAT
DOUBLE	DOUBLE PRECISION	DOUBLE	DOUBLE PRECISION
NUMERIC(p,s)	NUMERIC(p,s)		NUMERIC(p,s)
DECIMAL(p,s)	DECIMAL(p,s)		DECIMAL(p,s)
CHAR(n)	CHAR(n) n <= 255	CHAR(n) n <= 255	CHAR(n) n <= 32,767
VARCHAR(n)	VARCHAR(n) n <= 255	VARCHAR(n) n <= 255	VARCHAR(n) n <= 32,767
LONGVARCHAR	TEXT limit is 2 Gigabytes	LONGTEXT limit is 1.2 Gigabytes	LONG VARCHAR limit is 2 Gigabytes
BINARY(n)	BINARY(n) n <= 255	BINARY(n) n <= 255	BINARY n <= 32,767
VARBINARY	VARBINARY(n) n <= 255	VARBINARY(n) n <= 255	
LONGVARBINARY	IMAGE limit is 2 Gigabytes	LONGBINARY limit is 1.2 Gigabytes	IMAGE limit is 2 Gigabytes
DATE		DATE	DATE
TIME		TIME	TIME
TIMESTAMP	DATETIME		TIMESTAMP

Tabela 1: Tipos JDBC Types Mapeados para tipos SQL específicos dos Bancos de Dados

Para todos os métodos getXXX, pode ser fornecido ou o número da coluna ou o nome do campo. Por exemplo, para obter os nomes dos empregados no resultado da consulta, poderíamos tanto usar rs.getString (1) como rs.getString("nome").

Quando o tipo do método getXXX não corresponder ao tipo da coluna, Java tentará fazer conversões. Por exemplo, rs.getString ("salário") retorna o salário dos empregados como uma String.

6.6 PreparedStatement

Em um banco de dados, você pode fazer muitas consultas semelhantes que se difere em um ou dois parâmetros. Por exemplo, digamos que você deseja atribuir as notas de vários alunos cadastrados em uma tabela. A consulta UPDATE vai ser sempre a mesma, mudando apenas os parâmetros de nome e nota de cada estudante. Nesses casos, é possível utilizar um PreparedStatement. Primeiro você cria a instrução UPDATE geral e depois define os parâmetros para cada registro com o método setXXX. A vantagem é que o PreparedStatement é pré-compilado e, por isso, apresenta ganho em performance.

```
String comando = "UPDATE Estudantes SET nota = ? where name=?";
PreparedStatement ps = conn.prepareStatement (comando);
ps.setString (1, nota);
ps.setString (2, name);
int nroRegistrosAfetados  = ps.executeUpdate();
```

O comando PreparedStatement é útil quando é necessário criar uma instrução SQL muito complexa e tiver de executá-la várias vezes, modificando apenas os parâmetros.

Assim, como o Statement, o PreparedStatement possui os métodos executeQuery() para consultas SELECT e executeUpdate() para alterações dos registros das tabelas.

É possível obter metadados sobre os resultados das consultas contidos em um objeto ResultSet, através do objeto ResultSetMetaData. Este objeto é obtido pela chamada do método getMetaData de um ResultSet. Os metadados mais usados são:

getColumnCount (): retorna o número de colunas do resultado da consulta
getColumnLabel(int i): retorna o nome sugerido para título da coluna em impressão
getColumnName(in i): retorna o nome da coluna na tabela do banco de dados.

O método abaixo recebe o resultado de uma consulta (ResultSet) e imprime os seus dados.

```
private static void apresentaResultado(ResultSet rs) throws SQLException {
  int i;
  ResultSetMetaData rsmd = rs.getMetaData();
  // Pega o número de colunas da consulta
  int numCols = rsmd.getColumnCount();
  // Imprime o cabeçalho de cada coluna
  for(i=1; i<=numCols; i++)  {
    if(i > 1) System.out.print(",");
    System.out.print(rsmd.getColumnLabel(i));
  }
  System.out.println();
  // imprime o resultado da consulta
  while(rs.next()) //próximo registro {
    for (i=1; i<=numCols; i++)  {
      if(i > 1) System.out.print(",");
      System.out.print(rs.getString(i)); // imprime cada campo
    }
    System.out.println();
  }
}
```

6.7 Recuperando Exceções

JDBC permite que você veja warnings e exceções geradas pelo seu Banco de Dados e pelo compilador Java. Para ver exceções você precisa tratar o código com o bloco try/catch, como mostrado no capítulo de exceções.

As exceções em JDBC são objetos do tipo SQLException. Esse objeto é formado por três partes: a mensagem (uma string que descreve o erro), o estado SQL (uma string identificando o erro de acordo com as convenções SQLState X/Open) e o código de erro do driver (um valor int que normalmente é o código de erro do banco de dados). Para descobrir mais sobre o código de erro do driver você tem que consultar a documentação específica do driver.

Além disso, o bloco catch pode capturar mais de uma exceção SQL encadeadas, para fornecer maiores informações sobre um determinado erro que tenha acontecido.Para ver a próxima exceção, chame o método getNextException( ) da exceção corrente.

O exemplo abaixo demonstra um código com o bloco try/catch de tratamento de exceção SQLException.

```
import java.sql.*;
public class JdbcAccess {
  public static void main(String args []) {
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch(ClassNotFoundException e) {
      System.out.println("Não Consigo carregar o driver!!!");
      e.printStackTrace();
      return;
    }
       try {
          Connection conn = DriverManager.getConnection("jdbc:odbc:CursoJava", "",
"");
          Statement stmt = conn.createStatement();
          String comando = "SELECT * FROM Empregados";
          ResultSet rs = stmt.executeQuery(comando);
          while (rs.next()) {
              String nome = rs.getString (1);
              double salario = rs.getDouble ("SALARIO");
              System.out.println (nome.trim()+" "+salario);
          }
          stmt.close();
          conn.close();
       }
       catch (SQLException ex) {
              System.out.println ("Aconteceu uma exceção SQLException:");
              while (ex != null) {
                     System.out.println ("SQLState: "     + ex.getSQLState   ());
                     System.out.println ("Message: "      + ex.getMessage    ());
                     System.out.println ("Código do BD: " + ex.getErrorCode ());
                     ex = ex.getNextException ();
                     System.out.println ("");
              }
       }
  }
}
```

6.8 Recuperando Warnings

Objetos SQLWarning são subclasse de SQLException que lidam com advertências (warnings) de acesso ao banco de dados. Warnings não param a execução da aplicação, como uma exceção faz, eles apenas alertam o usuário de que alguma coisa não saiu como planejado. Por exemplo, um warning pode mostrar que aconteceu um erro durante uma tentativa de desconexão.

Um warning pode acontecer para os objetos Connection, Statement, PreparedStatement, CallableStatement ou ResultSet. Cada uma destas classes contém um método getWarnings( ), o qual é invocado para mostrar o primeiro warning. Se este método retornar um SQLWarning, pode ser invocado o método getNextWarning( ) deste objeto para serem obtidos os próximos warnings. Ao executar uma consulta, automaticamente são limpos os warnings das consultas anteriores. Por isso, para visualizar os warnings de uma consulta, é necessário fazer isso antes de executar outra consulta.

```
          while (rs.next()) {
              String nome = rs.getString (1);
              double salario = rs.getDouble ("SALARIO");
              System.out.println (nome.trim()+" "+salario);
              // A cada nova linha lida do ResultSet, os warnings são limpos
```

```
            SQLWarning warning = rs.getWarnings();
            if (warning != null) {
                  System.out.println ("----- Warnings ----------");
                   while (warning != null) {
                         System.out.println ("Message: " + warning.getMessage());
                         System.out.println ("SQLState: " +
warning.getSQLState());
                         System.out.println ("Código de erro: " +
warning.getErrorCode());
                         warning = warning.getNextWarning ();
                   }
            }
         }
         // A cada nova consulta, os warnings são limpos
         SQLWarning warning = stmt.getWarnings();
         if (warning != null) {
              System.out.println ("----- Warnings ----------");
            while (warning != null) {
                  System.out.println ("Message: " + warning.getMessage());
                  System.out.println ("SQLState: " + warning.getSQLState());
                  System.out.println ("Código de erro: " +
                                      warning.getErrorCode());
                  warning = warning.getNextWarning ();
            }
}
```

7 Criando Pacotes (Packages)

Java permite agrupar as classes em uma coleção chamada pacote. Isto permite organizar o trabalho e separar o código desenvolvido pelo programador do código de outras bibliotecas.

A biblioteca de classes Java é dividida em pacotes. Por exemplo, quando você usa:

```
import java.util.*;
```

na verdade, você está indicando que está usando classes do pacote java.util no seu código. O * é usado para indicar que pode ser qualquer classe deste pacote.

Para criar um pacote, você deve colocar o nome do pacote no topo do arquivo fonte. Este tem que ser o primeiro comando do arquivo, antes de algum import. Por exemplo, digamos que você queira criar um pacote cursojava.exemplos:

```
package cursojava.exemplos;
import java.util.*;
...
```

Você pode utilizar as classes em um pacote de duas maneiras, referenciando todo o nome do pacote ou importando o pacote. Digamos que se queira usar a classe Rectangle que pertença ao pacote cursojava.exemplos:

Primeiramente você pode fornecer o endereço do pacote:

```
cursojava.exemplos.Rectangle r = new cursojava.exemplos.Rectangle ();
```

Ou você pode importar o pacote e usar apenas o nome da classe:

```
import cursojava.exemplos.*;
Rectangle r = new Rectangle ();
```

Todos os arquivos de um pacote devem estar localizados em um subdiretório que corresponda ao nome do pacote. Por exemplo, as classes do pacote cursojava.exemplos devem estar no subdiretório cursojava\exemplos.

Esse subdiretório pode estar contido em qualquer diretório definido pelo usuário, porém esse diretório tem que estar definido na variável de ambiente CLASSPATH. Por exemplo, digamos que o subdiretório cursojava.exemplos esteja do diretório c:\unisinos. O CLASSPATH deve ser definido como:

```
CLASSPATH=.;c:\jdk\lib;c:\unisinos;
```

Para compilar os arquivos que contém a declaração package, deve ser usado o comando:

```
javac –d c:\unisinos Rectangle.java
```

Onde c:\unisinos é o diretório que vai conter os pacotes. A declaração –d força a criação do subdiretório cursojava\exemplos dentro do diretório c:\unisinos e coloca o arquivo ".class" dentro deste subdiretório. O diretório unisinos já deve existir.

7.1 Exercícios

Exercício 1: Jogo de adivinhação. Adaptado de (Santos, 2003).

Crie uma classe Java que se chama JogoDeAdivinhacao. A idéia é que o usuário tente adivinhar um número inteiro aleatório gerado automaticamente pelo programa. Esta classe contém uma constante chamada VALOR_MAXIMO que determina a faixa de valores do número a ser gerado (por exemplo, se o VALOR_MAXIMO é 100 o programa deverá gerar um número entre 0 e 100) e uma constante NRO_TENTATIVAS que representa o número máxima de tentativas permitidas ao usuário. Outro atributo é o valor gerado automaticamente pelo programa. Essa classe contém também um método *tenta ( )* que durante *NRO_TENTATIVAS* vezes lê os valores fornecidos pelo usuário e verifica se o usuário adivinhou. Caso afirmativo, gera uma mensagem de parabenização ao usuário. Se o usuário não adivinhar o número após NRO_TENTATIVAS vezes, o programa notifica o usuário e finaliza a execução.

Crie a classe TestaAdivinhação que possui o método main. No método main crie um objeto do tipo JogoDeAdivinhacao e execute o método *tenta ( )* para iniciar o jogo.

Dicas: Use a classe Random do pacote java.util para a geração de números aleatórios. O Número aleatório deve ser gerado no método construtor.

```
Random r = new Random ();
valor = r.nextInt(VALOR_MAXIMO);
```

Use a classe Console do pacote cursojava.io (fornecido pela professora) para ler um número fornecido pelo usuário através do teclado.

8 Arquivos JAR

Um arquivo JAR (Java Archive) é um arquivo compactado produzido por uma ferramenta de mesmo nome. Um arquivo JAR pode conter não somente classes Java (*.class*), mas também, vários arquivos diferentes, tais como, imagens, sons de áudio e outros. Além de serem compactados pela ferramenta JAR, esses arquivos também podem ser compactados (ou descompactados) usando qualquer ferramenta de compressão de formato ZIP.

A ferramenta JAR é bastante útil para compactação dos arquivos necessários para exibição de um applet (classes, imagens, áudio), tornando o processo de exibição do applet mais rápido. Por exemplo, quando o browser carrega uma página HTML que contém o código para executar um applet Java, o browser faz uma conexão ao servidor Web e copia para a máquina cliente a classe do applet. Quando o interpretador Java (embutido no browser) carrega a classe do applet Java, ele verifica que são necessárias mais quatro classes para executar o applet. Assim, o browser faz mais quatro conexões para o servidor Web, uma para cada arquivo de classe. Isso pode ser um processo de consumo de tempo desnecessário. Uma solução é colocar todos os arquivos compactados dentro de um arquivo JAR para serem copiados em uma única conexão pelo browser.

8.1 A Ferramenta JAR

A ferramenta JAR, que acompanha o JDK, possui as seguintes opções:

Opção	Descrição
c	Cria um novo arquivo na saída padrão
t	Lista o conteúdo do arquivo na saída padrão
f	Especifica o nome do arquivo a ser criado, listado ou de onde os arquivos serão extraídos
x	Estrai os arquivos especificados do arquivo JAR
0 (Zero)	Indica que os arquivos não deverão ser comprimidos, apenas incluídos no arquivo
m	Indica que um arquivo de manifesto externo deve ser incluído jar cmf Manifest.mf arq.jar *.class
M	Indica que um arquivo de manifesto externo não deve ser incluído

Para ilustrar o uso da ferramenta JAR, descrevemos a seguir alguns exemplos.

8.1.1 Para criar um arquivo JAR:

```
jar cf arquivoJar.jar *.class
jar cvf arquivosJar.jar Ex1.class Diretorio1 v -> mostra passos na tela
```

8.1.2 Para visualizar o conteúdo de um arquivo JAR:

```
jar tf arquivoJar.jar
```

8.1.3 Para extrair o conteúdo de um arquivo JAR:

```
jar xf arquivoJar.jar -> (Extrai todos)
jar xf arquivoJar.jar arq1.class -> (Extrai apenas arq1.class)
```

Quando o programa Jar extrai arquivos, ele sobrescreve os arquivos que tenham o mesmo nome e caminho que os arquivos que estão sendo extraídos.

O arquivo JAR fica intacto.

8.1.4 Para executar uma aplicação armazenada em um arquivo JAR

(version 1.2 - requer especificação da classe Main no arquivo de Manifest)

```
java -jar app.jar
```

8.1.5 Para invocar um applet armazenado dentro de um arquivo JAR:

```
<applet code=AppletClassName.class
        archive="JarFileName.jar"
        width=width height=height>
</applet>
```

8.2 Arquivo de Manifesto

O arquivo de manifesto serve para guardar meta informações sobre os arquivos que estão armazenados em um arquivo JAR. Este arquivo é automaticamente gerado pela ferramenta JAR com o *pathname* META-INF/MANIFEST.MF dentro do arquivo .jar.

Quando criado automaticamente pela ferramenta JAR, o arquivo de manifesto possui as seguintes informações:

```
Manifest-Version: 1.0  // obedece a versão 1.0 da especificação de manifestos
Created-By: 1.3.0 (Sun Microsystems Inc.) // versão do JDK
```

As entradas do arquivo de manifesto possuem o formato:

```
"header: valor"
```

Por exemplo: Main-Class: Cliente

A entrada "Main-Class: Cliente" tem como objetivo indicar qual classe contém o método main:

```
public static void main (String args[ ])
```

por onde a MVJ deve iniciar a execução. Ela é requerida a partir da versão 1.2 do JDK para que o interpretador possa executar diretamente uma aplicação compactada dentro de um arquivo JAR. Por exemplo:

```
java -jar app.jar
```

Na linha acima, estamos executando a aplicação contida dentro do arquivo app.jar. Através das informações do arquivo de manifesto, o interpretador sabe por qual classe ele deve iniciar a execução.

8.2.1 Para criar arquivo JAR com um dado manifesto:

```
jar cmf infoManifesto.txt  app.jar *.class
```

8.2.2 Para adicionar informações ao manifesto de um arquivo JAR

```
jar umf infoManifesto.txt  app.jar
```

onde infoManifesto.txt é um arquivo texto que contém as informações (entradas) a serem adicionadas ao manifesto.

9 Servlets

Java Servlet é um componente chave no desenvolvimento de aplicações Java para servidores (server-side). Um servlet é uma pequena e plug-in extensão para um servidor e que, por isso, herda as suas funcionalidades. Servlets permitem estender as funcionalidades de qualquer servidor que entenda Java (como um servidor Web ou um servidor de e-mail), porém são mais utilizados para estender servidores Web, fornecendo uma substituição eficiente aos *scripts* CGI (Common Gateway Interface). Por exemplo, um servlet pode ser responsável por receber dados de um formulário HTML, realizar uma consulta em um banco de dados e exibir uma resposta em uma página HTML.

Um servlet executa dentro de uma Máquina Virtual Java (Java Virtual Machine - JVM) no servidor. Por isso, os servlets operam dentro do domínio do servidor: ao contrário dos applets, eles não requerem suporte no browser, ou seja, não é necessário que o browser reconheça aplicações Java.

Ao contrário de CGIs que usam múltiplos processos para gerenciar as diferentes requisições (cada requisição HTTP é um novo processo CGI), servlets são *threads* dentro do servidor web, conforme ilustra Figura 4. Como servlets executam dentro de um servidor Web, eles permitem realizar ações que não poderiam ser feitas por um CGI. Por exemplo, um script CGI não pode escrever no arquivo de log do servidor.

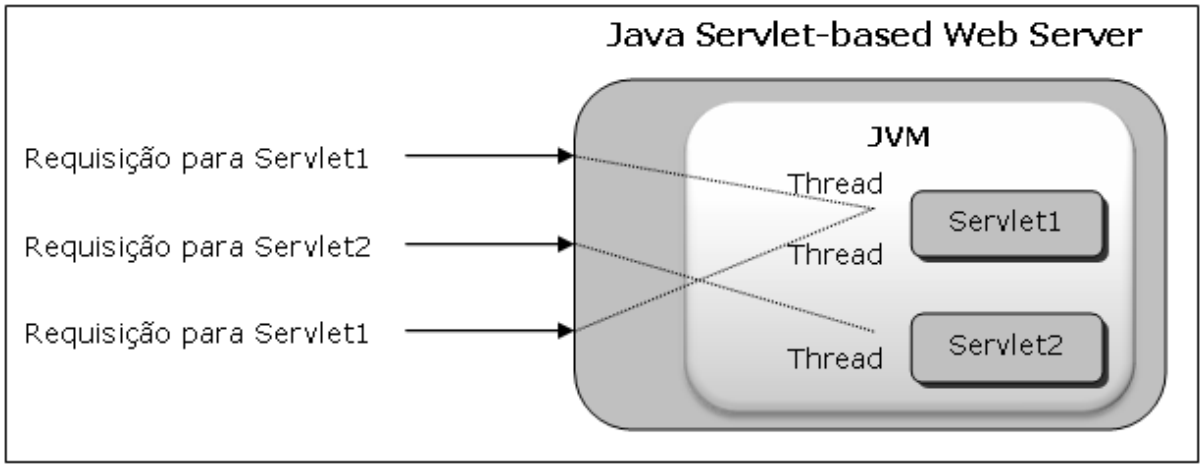

Figura 4: Ciclo de Vida de um Servlet

9.1 Suporte para Servlets

Os servlets são portáveis, ou seja, assim como o Java são independentes de plataforma. Para desenvolver servlets é necessário um conjunto de classes que fornecem suporte básico a servlets. Os pacotes javax.servlet e javax.servlet.http constituem a Servlet API. Embora os servlets façam parte da linguagem Java, esses pacotes não são oferecidos como parte da Java API (JDK). Anteriormente, a API servlet encontrava-se disponível em um outro pacote denominado Java Servlet Development Kit (JSDK)[3], para ser usado juntamente com o JDK. Atualmente, a API para servlets faz parte do pacote Java 2SDK, Enterprise Edition (JS2EE)[4] que se encontra disponível na versão 1.2.1. Além da API para servlets, o JS2EE inclui pacotes para suporte a JavaServer Pages, JavaBeans, Java Servlets e XML.

A API de servlets é necessária para você criar e compilar os seus servlets. Porém, para executar e testar um servlet é necessário um Servlet Engine. A escolha da Servlet Engine vai depender do servidor web usado. Eles podem ser stand-alone ou add-on.

Um *Stand-alone Servlet Engine* (SSE) é um servidor web que possui suporte para servlets. A desvantagem de utilização de um SSE é a necessidade de esperar uma nova versão do servidor para ter a última versão de servlets. Um exemplo de Stand-alone Engine é o servidor TomCat do projeto Jakarta[5]. Um outro exemplo é o Resin Server da Caucho[6].

Um *Add-on Servlet Engine* (ASE) funciona como um *plug-in* a um servidor existente – ele adiciona suporte para servlets a um servidor que originalmente não tinha sido construído para suportar servlets. O Servidor Tomcat também executa como um módulo do servidor web Apache e do Internet Information Server, entre outros.

Para aprender a instalar o Apache Tomcat versão 6.0 como um *Stand-alone Servlet Engine*, veja a página http://tomcat.apache.org/tomcat-6.0-doc/setup.html ou ainda: http://java.sun.com/developer/onlineTraining/JSPIntro/exercises/SetupTomcat/index.html.

9.2 Configurando o TomCat como um Add-on Servlet Engine

Uma vez que o servlet tenha sido escrito, você pode testá-lo com o TomCat.

O Jakart TomCat encontra-se na versão 4.0 (Beta). Trata-se de um servlet engine desenvolvido no projeto Jakarta da Apache que utiliza a Servlet API 2.3. O TomCat pode ser usado tanto como um stand-alone servlet e JSP engine como também um plug-in para os servidores:

Apache, version 1.3 ou posterior
Microsoft Internet Information Server, version 4.0 ou posterior
Microsoft Personal Web Server, version 4.0 ou posterior
Netscape Enterprise Server, version 3.0 ou posterior.

---

[3] *O JSDK ainda encontra-se disponível para download na página http://java.sun.com/products/servlet/download.html.*

[4] *O J2SDK, Enterprise Edition encontra-se disponível para download no site http://java.sun.com/j2ee/download.html.*

[5] *http://jakarta.apache.org/*

[6] *http://www.caucho.com/*

Passos para instalação e configuração do TomCat:

1) Copie o TomCat do endereço: http://jakarta.apache.org/tomcat/

2) Descompacte o arquivo na raiz do C. Ele criará um diretório C:\jakarta-tomcat\.

3) Inclua as variáveis de ambiente necessárias nos arquivos de script do TomCat. Estes arquivos são o startup.bat e shutdown.bat e estão nos diretórios C:\jakarta-tomcat\bin. Por exemplo, na minha máquina o Java está instalado no diretório C:\JDK1.3, então nos arquivos startup.bat e shutdown.bat, após a linha:

```
if not "%TOMCAT_HOME%" == "" goto start
```

Adicione as linhas:

```
SET JAVA_HOME=C:\JDK1.3
SET TOMCAT_HOME=C:\jakarta-tomcat
```

4) Modifique o Classpath no arquivo autoexec.bat:

Adicione os seguintes caminhos a variável de ambiente classpath:

C:\jakarta-tomcat\lib\servlet.jar
C:\jakarta-tomcat\lib\jasper.jar

O diretório onde estão instalados os seus diretórios de pacote.

5) Inicie o TomCat:

Dê um duplo clique em startup.bat. Você pode criar um shortcut (um atalho) para startup.bat e shutdown.bat em seu diretório de trabalho ou no desktop, o que torna mais fácil iniciar e finalizar o TomCat. Se você obtiver uma mensagem de erro "Out of Environment Space" ao iniciar o TomCat, então dê um clique direito no startup.bat, selecione Properties, selecione Memory, e então ache a entrada "Initial Environment". Modifique o valor para 2816 e clique no botão "OK". Repita a operação para o arquivo shutdown.bat.

6) Teste o Servidor

Abra o browser e digite http://localhost:8080. Será aberta uma página do TomCat.

7) Compile e instale as suas classes de servlets:

Os arquivos .class vão para o diretório C:\jakarta-tomcat\webapps\ROOT\WEB-INF\classes ou C:\jakarta-tomcat\webapps\ROOT\WEB-INF\classes\seuPacote. Você pode usar a opção –d do compilador para os arquivos .class serem gerados neste diretório. Uma outra opção é você setar a variável de ambiente classpath para apontar para o diretório em que estão as suas classes de servlet.

8) Teste seus servlets:

Digite a URL http://localhost:8080/servlet/NomeDoServlet ou http://localhost:8080/servlet/seuPacote.NomeDoServlet. Onde NomeDoServlet é o nome da classe (.class).

9) Instale os arquivos HTML:

Os arquivos vão para o diretório C:\jakarta-tomcat\webapps\ROOT ou C:\jakarta-tomcat\webapps\ROOT\seuDiretorio.

9.3 Interagindo com o Cliente

Quando um cliente (navegador web) se conecta a um servidor para fazer uma requisição HTTP, esta requisição pode ser de vários tipos, que são chamados métodos. Os métodos mais usados são o GET e o POST. O método GET é usado para obter informações (um documento ou os resultados de uma consulta a um banco de dados), enquanto o POST é usado para o envio de informações (como o número de cartão de crédito ou um dado para ser armazenado em um banco de dados).

Na requisição GET as informações solicitadas são enviadas como uma seqüência de caracteres adicionada a URL de requisição, o que é chamado *query string*. Colocar a informação na URL permite que a URL seja armazenada em um *Bookmark* para posterior acesso. Alguns servidores limitam o tamanho da URL e de *query strings* em 240 caracteres.

O método POST usa uma técnica diferente para enviar informações ao servidor, pois em alguns casos é necessário enviar até megabytes de informações. A requisição passa todos os dados através da conexão *socket*[7], como parte do corpo da requisição HTTP. A troca de informações é transparente para o cliente. A URL não é modificada e, por isso, requisições POST não podem ser gravadas no *bookmark*.

Servlets usam classes e interfaces de dois pacotes: *javax.servlet* e *javax.servlet.http*. O pacote *javax.servlet* contem classes para suportar servlets genéricos. Estas classes estendem as classes do pacote *javax.servlet.http* para possuir funcionalidades HTTP específicas.

Todo o servlet HTTP deve derivar da classe HttpServlet que é subclasse de GenericServlet. Ao contrário de um programa Java comum, um servlet não tem uma função *main( )* por onde é iniciada a execução. Ao invés disso, certos métodos de um servlet são chamados pelo servidor quando este recebe as requisições. Toda vez que um servidor dispara uma requisição para um servlet, ele dispara o método service( ) do servlet. Esse método dispara o método apropriado para tratar a requisição feita que pode ser doGet( ) ou doPost( ). Estes métodos devem ser sobrescritos na implementação do servlet e tratam as requisições do tipo GET, doGet( ), e POST, doPost( ).

Os métodos **doGet** e **doPost** recebem como parâmetros dois objetos:

Um objeto HttpServletRequest, a qual encapsula os dados vindos do cliente;
Um objeto HttpServletResponse, a qual encapsula a resposta ao cliente.

O objeto HttpServletRequest fornece acesso aos dados de header HTTP, tais como cookie e permite obter os argumentos que o usuário enviou como parte da requisição.

Para acessar esses parâmetros existem os métodos:

getParameter (String nome_parâmetro): retorna o valor o parâmetro de nome dado;
getParameterValues (String nome_parâmetro): Se o parâmetro possui mais de um valor. Este método retorna uma array dos valores do parâmetro.
getParameterNames ( ): retorna o nome dos parâmetros.

---

[7] *Um socket é a extremidade de uma conexão, de duas direções, entre dois programas executando em rede. Um socket possui um número de porta para identificar para qual das aplicações no computador, o dado deve ser enviado (Sun, 2007).*

Um objeto HttpervletResponse fornece duas maneiras de retornar dados ao usuário:

- O método getWriter( ) retorna um objeto PrintWriter. Usa-se o método getWriter( ) para retornar dados textuais ao usuário.
- O método getOutputStream( ) retorna um servletOutputStream( ). Usa-se o método para dados binários.

É necessário determinar o tipo do conteúdo de retorno. Para tanto, usa-se o método setContentType( ) do objeto HttpServletResponse. Para retornar páginas HTML, especifique o conteúdo como *"text/html"*.

9.4 Gerando uma página HTML no cliente

Uma das funções mais simples que podemos querer fazer com um servlet é gerar dinamicamente uma página web. Para tanto, precisamos apenas sobrescrever o método doGet(...) da classe HttpServlet, que trata a requisição get da página, na classe servlet que criamos. Esse método recebe como parâmetro um objeto HttpServletResponse, o qual representa a informação a ser enviada ao usuário quando o servlet for chamado. Através deste objeto, chamando o método response.getWriter(), podemos obter o objeto PrintWriter que é usado como stream de saída pelo servlet. Veja o exemplo de código abaixo:

```
1.import java.io.*;
2.import javax.servlet.*;
3.import javax.servlet.http.*;
4.public class SimpleServlet extends HttpServlet {
5.   public void doGet (HttpServletRequest request, HttpServletResponse response)
6.     throws ServletException, IOException {
7.     // seta o tipo de conteúdo
8.     response.setContentType("text/html");
9.     String title = "Simple Servlet Output";
10.    // então escreve os dados de resposta
11.    PrintWriter  out = response.getWriter();
12.    out.println("<HTML><HEAD><TITLE>");
13.    out.println(title);
14.    out.println("</TITLE></HEAD><BODY>");
15.    out.println("<H1>" + title + "</H1>");
16.    out.println("<P>This is output from SimpleServlet.");
17.    out.println("</BODY></HTML>");
18.    out.close();
19.   }
20.}
```

No código acima, nas linhas 1 a 3 estamos importando os pacotes Java que serão utilizados neste código (java.io para entrada e saída, e javax.servlet e javax.servlet.http que são as API para servlets). Da linha 5 a 19, estamos sobrescrevendo o método doGet(...) da classe pai para responder a requisições do tipo GET realizadas a este servlet. Na linha 8, estamos definindo que a saída será uma página HTML e na linha 11 estamos obtendo o objeto PrintWriter que é usado como stream de saída pelo servlet. O método println() deste objeto escreve a String passada como parâmetro na página HTML de saída. Como o resultado é uma página HTML, é importante colocar as tags HTML necessárias nesta String de saída.

O servlet acima gera a seguinte saída no navegador web do usuário:

Figura 5: Saída do Servlet "Simple Servlet Output"

9.5 Recebendo parâmetros do cliente

O exemplo anterior apenas gera uma página HTML no cliente, ou seja, no navegador web que fez a requisição. Vejamos agora como obter os dados enviados por um formulário HTML.

Figura 6: Formulário que chama o servlet "Pega Parâmetro"

Por exemplo, esse é o código da página HTML que contém o formulário exibido na Figura 6:

```
1. <HTML>
2. <HEAD>
3. <TITLE>Servlet Pega Parâmetro</TITLE></HEAD>
4. <BODY>
5. <FORM METHOD="get"
6.        ACTION="http://localhost:8080/servlet/ServletParametro" >
7.       Nome:
8.       <INPUT TYPE="text" SIZE="50" NAME="nome">
9.       <INPUT TYPE=submit VALUE="Enviar">
10. </FORM>
11. </BODY>
```

Na linha estamos criando o formulário com a *tag* <FORM...>. Quando se deseja criar um formulário HTML que envie dados a um servlet, devemos determinar o método (POST ou GET), isso é informado pelo atributo METHOD da *tag*, e o nome e URL do servlet que irá tratar os dados (indicado pelo atributo ACTION da *tag*). Na linha 8, estamos criando uma caixa de texto de tamanho 50 com nome "*nome"* e por último a linha 9, cria um botão com rótulo "*Enviar*", que quando clicado pelo usuário irá submeter o dado fornecido pelo usuário na caixa de texto ao servlet especificado na linha 6.

O servlet para receber o parâmetro do formulário acima pode ser implementado da seguinte maneira:

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4. public class ServletParametro extends HttpServlet {
5.     public void doGet (HttpServletRequest  request, HttpServletResponse response)
6.                                          throws  ServletException, IOException {
7.            String nome = request.getParameter ("nome");
8.            // seta o tipo de conteúdo
9.            response.setContentType("text/html");
10.           PrintWriter  out;
11.           String title = "Simple Servlet Pega-Parâmetro";
12.           // então escreve os dados de resposta
13.           out = response.getWriter();
14.              out.println("<HTML><HEAD><TITLE>");
15.           out.println(title);
16.           out.println("</TITLE></HEAD><BODY>");
17.           out.println("<H1>" + title + "</H1>");
28.           out.println("<P><H4>Esta é uma página gerada pelo Servlet Pega-
19.                    Parâmetro. <br></H4><H3> Ola " + nome + "! :-)</H3>");
20.           out.println("</BODY></HTML>");
21.           out.close();
22.    }
23. }
```

Observe que como os dados são submetidos pelo método GET, as informações enviadas são tratadas no servlet pelo método doGet( ) (linha 5 do código acima).

Além disso, os dados do formulário são obtidos através do método getParamater (String campo) do objeto HttpServletRequest, como pode ser observado na linha 7 do código acima. A variável **campo** é uma String que contém o nome do campo de formulário.

O servlet acima gera a seguinte página HTML, caso o usuário tenha entrado o nome "Patrícia" no formulário acima:

Figura 7: Saída do servlet "Pega Parâmetro"

9.6 Gerenciando requisições POST

Os dois servlets implementados nas seções anteriores apenas tratavam requisições GET. Para fazer com que o ServletParametro trate requisições POST é necessário que se sobrescreva o método doPost(). Digamos que o método doPost() deva realizar a mesma função que o método doGet(). Neste caso, é necessário apenas chamar o método doGet() de dentro do método doPost().

```
1. public void doPost (HttpServletRequest req, HttpServletResponse res) throws
2.     ServletException, IOException  {
3.     doGet (req, res);
4. }
```

Assim, o servlet pode também tratar informações provenientes de um formulário que envie submissões através do método POST:

```
<FORM METHOD="post" ACTION="http://localhost:8080/servlet/ServletParametro">
```

9.7 O Ciclo de Vida de um Servlet

Um Servlet Engine executa todos os seus servlets em uma única Máquina Virtual Java (JVM). Por elas estarem em uma única JVM, servlets podem trocar dados entre si.

Um servlet tem o seguinte ciclo de vida:

Um servidor carrega e inicia o servlet. Quando um servidor carrega um servlet, o servidor roda o método *init* do servlet. Este método é chamado apenas quando o servlet é carregado;
O servlet trata zero ou mais requisições de clientes;
O servidor remove o servlet (geralmente, isto acontece apenas quando o servidor encerra a sua execução).

Quando um servlet é carregado, o servidor Web cria uma única instância para este servlet. Esta única instância gerencia todas as requisições solicitadas para este servlet.

Para entender melhor isso, veja o exemplo abaixo:

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4. public class SimpleCount extends HttpServlet {
5.     int count =0;
6.     public void doGet (HttpServletRequest request, HttpServletResponse response
7.                         ) throws ServletException, IOException {
8.        response.setContentType("text/html");
9.           PrintWriter         out;
10.          String              title = "Simple Servlet Output";
11.          out = response.getWriter();
12.         out.println("<HTML><HEAD><TITLE>");
13.          out.println(title);
14.          out.println("</TITLE></HEAD><BODY>");
15.          out.println("<H1>" + title + "</H1>");
16.          out.println("<P>Este é o "+ ++count + " acesso desse servlet!");
17.          out.println("</BODY></HTML>");
18.          out.close();
19.    }
20. }
```

O objetivo deste código é demonstrar que o servidor Web cria apenas uma instancia para cada servlet. Para cada chamada a um servlet, ou seja, cada usuário que acessa o servlet através do

seu navegador web, é criada uma thread que acessará aquela única instância de servlet. Por isso, se um usuário acessa um servlet e modifica o valor de uma variável membro, o valor dessa variável permanecerá modificado para todos os objetos que acessarem posteriormente. Isso é o que queremos demonstrar com a nossa variável de classe count.

Como um servlet é uma única instância que é chamada por vários threads, onde cada thread é uma requisição cliente, é possível que o servlet imprima o mesmo valor do contador (count) para duas requisições que tenham sido realizadas ao mesmo tempo.

Desta maneira, é interessante sincronizarmos[8] o bloco que não se deseja que seja executado ao mesmo tempo por outros threads (requisições). Para garantir que não seja impresso o mesmo valor do contador para duas requisições feitas ao mesmo tempo, o método doGet() deve ser sincronizado. O código ficaria como a seguir:

```
public synchronized void doGet (HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
```

9.8 Parâmetros de Inicialização de um Servlet

Assim como os *applets*, os servlets possuem os métodos init() e destroy(). O método init() é chamado automaticamente após a criação da instância do servlet. Dependendo da implementação do servidor web, isso é feito quando o servidor inicia a sua execução ou quando o servlet é requisitado pela primeira vez.

O método init() é tipicamente usado para iniciar o servidor. Por exemplo, se você deseja criar um servlet que acesse uma base de dados, a conexão ao banco de dados deve ser realizada no método init() do servlet. Além disso, informações do ambiente podem ser obtidas pelo servlet através do parâmetro do tipo ServletConfig que é recebido no método init( ).

Através do ServletConfig é possível especificar parâmetros de iniciação do servlet, mas que não estão associados a cada requisição. Como esses parâmetros de inicialização serão passados ao servidor vai depender da implementação do servidor. No *TomCat* esses parâmetros devem ser fornecidos no arquivo Web.xml que está localizado no diretório *C:|jakarta-tomcat|webapps|ROOT|WEB-INF*.

Por exemplo, digamos que queremos que um servlet bookdb leia dados do arquivo DataBase.txt. O arquivo teria que ser configurado da seguinte maneira:

```
<web-app>
    <servlet>
        <servlet-name>
            bookdb
        </servlet-name>
        <servlet-class>
            BookDB
        </servlet-class>
        <init-param>
            <param-name>dbfile</param-name>
            <param-value>DataBase.txt</param-value>
```

[8] *Um thread executando pode acessar qualquer objeto a qual ela tem referência. Desta maneira, é possível que dois threads tentem acessar o mesmo objeto, interferindo umas nas outras. Deve-se tomar o cuidado que os threads acessem o objeto um por vez. A sincronização consiste em definir a parte do código que deve ser acessada por um thread por vez.*

```
        </init-param>
    </servlet>
</web-app>
```

O parâmetro de inicialização descrito acima pode ser obtido através dos métodos do objeto ServletConfig obtido no método init( ):

Quando há um parâmetro:

```
String dbfile = config.getInitParameter("dbfile");
```

Vejamos um exemplo de um servlet inicializado. No programa abaixo o servlet SimpleCount foi modificado para receber um valor inicial para o contador. Isso é realizado na linha 8.

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4. public class InitCounter extends HttpServlet {
5.     int count =0;
6.     public void init (ServletConfig config) throws ServletException  {
7.            super.init (config);
8.            String valorInicial = config.getInitParameter ("valorInicial");
9.            try {
10.                  count = Integer.parseInt(valorInicial);
11.           }
12.           catch (NumberFormatException e)  {
13.                  count=0;
14.           }
15.    }
16.    public void doGet (HttpServletRequest  request, HttpServletResponse
17.           response) throws ServletException, IOException {
18.           response.setContentType("text/html");
19.           PrintWriter         out;
20.           String              title = "Simple Servlet Output";
21.           out = response.getWriter();
22.           out.println("<HTML><HEAD><TITLE>");
23.           out.println(title);
24.           out.println("</TITLE></HEAD><BODY>");
25.           out.println("<H1>" + title + "</H1>");
26.           out.println("<P>Este é o "+ ++count + " acesso desse servlet!");
27.           out.println("</BODY></HTML>");
28.           out.close();
29. }
30.}
```

O código do arquivo *web.xml* que contém a informação de iniciação obtida na linha 8 do código acima poderia ser descrito da seguinte maneira:

```
<web-app>
    <servlet>
        <servlet-name>
            contador
        </servlet-name>
        <servlet-class>
            InitCounter
        </servlet-class>
        <init-param>
            <param-name>valorInicial</param-name>
            <param-value>7</param-value>
        </init-param>
    </servlet>
</web-app>
```

9.9 Usando Cookies

Cookie é uma maneira de um servidor (ou um servlet, como parte de um servidor) enviar informações para o cliente (navegador web do usuário) armazenar. Servlets enviam *cookies* através do objeto HttpServletResponse e clientes retornam *cookies* através do objeto HttpServletResponse.

Os servlets que rodam dentro de um único servidor compartilham os *cookies*. Isso significa que um servlet pode guardar o valor de um *cookie* e outro servlet o requisitar.

9.9.1 Criando um Cookie

O valor de um cookie pode ser qualquer String sem espaço em branco e sem os caracteres: [ ] ( ) = , " / ? @ : ;

O cookie deve ser criado antes de acessar o objeto PrintWriter, já que os *cookies* são enviados ao usuário como *header* e *headers* devem ser enviados ao usuário antes de acessar o objeto PrintWriter. Após criar o *cookie* (linha 8), ele deve ser enviado ao cliente usando o método addCookie() do objeto HttpServletResponse (linha 9).

```
1.import java.io.*;
2.import javax.servlet.*;
3. import javax.servlet.http.*;
4. public class ServletGravaCookie extends HttpServlet {
5.   public void doGet (HttpServletRequest request, HttpServletResponse response)
6.                                    throws ServletException, IOException {
7.     response.setContentType("text/html");
8.       Cookie cookie = new Cookie ("Cookie", "deucerto");;
9.       response.addCookie(cookie);
10.      PrintWriter            out;
11.      String title = "Servlet Grava Cookie";
12.     out = response.getWriter();
13.     out.println("<HTML><HEAD><TITLE>");
14.      out.println(title);
15.      out.println("</TITLE></HEAD><BODY>");
16.      out.println("<H1>" + title + "</H1>");
17.      out.println("<P>Gravou cookie.");
18.      out.println("</BODY></HTML>");
19.      out.close();
20.   }
21. }
```

9.9.2 Recuperando Cookies

Um *cookie* pode ser recuperado através do método getCookies() da classe HttpServletRequest. Este método retorna um *array* de objetos *cookies* (linha 8 no código abaixo), na qual você deve procurar o *cookie* desejado através de seus nomes identificadores (linha 11). Para obter o nome do *cookie* use o método getName() (linha 11) e para o obter o seu valor use o método getValue() (linha 12).

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4. public class ServletPegaCookie extends HttpServlet {
5.     public void doGet (HttpServletRequest  request, HttpServletResponse
6.           response) throws ServletException, IOException {
7.        String valorCookie = null;
8.       Cookie[] cookies = request.getCookies();
9.      for(int i=0; i < cookies.length; i++)  {
10.               Cookie thisCookie = cookies[i];
```

```
11.                if (thisCookie.getName().equals("Cookie"))
12.                        valorCookie = thisCookie.getValue();
13.        }
14.        PrintWriter out;
15.        String    title = "Servlet Pega Cookie";
16.        // set€a o tipo do conteúdo
17.            response.setContentType("text/html");
18.        // então escreve para o cliente
19.        out = response.getWriter();
20.          out.println("<HTML><HEAD><TITLE>");
21.        out.println(title);
22.        out.println("</TITLE></HEAD><BODY>");
23.        out.println("<H1>" + title + "</H1>");
24.        out.println("<P>Pegou cookie de nome Cookie e Valor="+valorCookie);
25.        out.println("</BODY></HTML>");
26.        out.close();
27.     }
28. }
```

9.10 Redirecionando uma Requisição

Um servlet pode direcionar uma requisição para uma página HTML, ou seja, ele envia informações para o cliente exibir uma determinada página HTML. O exemplo abaixo realiza um redirecionamento randômico para alguns sites.

```
1. import java.io.*;
2. import javax.servlet.*;
3. import javax.servlet.http.*;
4. import java.util.*;
5. public class SiteSelector extends HttpServlet {
6.     Vector sites = new Vector ();
7.     public void init (ServletConfig config) throws ServletException  {
8.            super.init (config);
9.            sites.add ("http://www.oreilly.com/catalog/jservlet/");
10.           sites.add ("http://www.servlets.com");
11.           sites.add ("http://jserv.java.sun.com");
12.           sites.add ("http://www.servletcentral.com");
13.    }
14.    public void doGet (HttpServletRequest request,  HttpServletResponse response)
15.     throws ServletException, IOException {
16.           int siteIndex = (int)(Math.random()*4);
17.           String site = (String) sites.elementAt (siteIndex);
18.           response.sendRedirect(site);
19.    }
20. }
```

No código acima, na linha 6 estamos criando uma lista (com a classe Vector do pacote java.util.* do java) e adicionando nesta lista URLs, como String, de *sites* web (linha 9 a 12). No método doGet(), que trata requisições do tipo GET do cliente, na linha 16 é gerado um número randômico entre 0 e 3 (o método Math. Random() gera randomicamente um número em ponto flutuante com valor entre 0 e 1, exclusive. Ao multiplicar por 4, estamos garantindo que seja que seja obtido um número inteiro, por causa do *casting* explícito, entre 0 e 3). Na linha 17, é obtida a URL armazenada naquela posição da lista e, finalmente, na linha 18 é realizado um redirecionamento para a URL obtida na linha anterior.

10 Java Server Pages

A tecnologia JavaServer Pages (JSP) permite colocar pedaços de código de servlets em um arquivo textual. Uma página JSP é um documento textual que contém 2 tipos de texto: dado estático (que pode ser expresso em qualquer formato baseado em texto como HTML e XML) e elementos JSP, que determinam como o conteúdo dinâmico da página é construído. Você pode escrever o código HTML usando a ferramenta de autoria de sua preferência. Então, você inclui o código Java para a parte dinâmica em tags especiais, em sua grande parte começando com "<%" e finalizando com "%>".

As páginas JSP são usadas nas aplicações web em conjunto com os servlets, sendo que as primeiras são mais voltadas a preencher a função de interface com o usuário, já que permitem misturar código Java com texto HTML formatado. Os arquivos JSP geralmente possuem a extensão *.jsp*, como em *mypage.jsp*.

Quando um cliente (um usuário acessando uma página JSP através de seu navegador web) faz uma requisição a uma página JSP pela primeira vez, o servidor web traduz a página JSP para um servlet. Nas próximas vezes que a página JSP for chamada, é o servlet que é chamado. Essa fase de tradução da página JSP para um servlet é transparente ao programador e ao usuário.

O JSP não oferece funcionalidades a mais que não se possa conseguir com um servlet puro. Entretanto, ele oferece a vantagem de ser mais intuitivo para programação, facilitando assim a elaboração e manutenção da página dinâmica. Além disso, por ser um documento textual, a página JSP permite que os programadores trabalhem na parte dinâmica (código Java) deixando a apresentação gráfica da página em HTML para *designers*.

10.1 Elementos JSP

Elementos JSP de *scripting* permitem inserir o código Java na página JSP. Lembre-se que como essa página poderá conter também código HTML, são necessárias *tags* especiais indicando que aquele determinado trecho possui código Java que precisa ser executado no servidor. Essas *tags* especiais são os elementos JSP de scripting.

Expressões na forma de <%= expressão %> que são avaliadas e inseridas na saída,
Scriptlets na forma <% code %> que são inseridos no método service do servlet,
Declarações na forma <%! code %> que são inseridas no corpo da classe servlet, fora de qualquer método.

Vejamos nas próximas seções cada um destes tipos de tag.

10.1.1 Expressões

Vamos ver um exemplo de um JSP (retirado de baseado em *http://www.exampledepot.com/egs/javax.servlet.jsp/getparam.html*) com tag de expressão:

```
<html>
<body>
<h1><%= request.getParameter("name") %></h1>
</body>
</html>
```

Se você salvar esse JSP com o nome "*teste.jsp*", e a página for acessada com a URL:

http://hostname.com/mywebapp/mypage.jsp?name=Patricia+Jaques

A saída resultante seria:

```
Hello <b>Patricia Jaques</b>!
```

Isso porque estamos fazendo uma requisição GET e nesse tipo de requisição os parâmetros são obtidos da própria string da requisição (o dado contido após o ponto de interrogação). No exemplo acima a URL está passando o dado "*Patricia Jaques*" como valor para o parâmetro *name*.

Se o valor do parâmetro não fosse passado, a saída resultante seria:

```
Hello <b>null</b>!
```

Quando a página é requisitada, a expressão é avaliada, convertida em uma string e inserida na página.

Um outro exemplo de expressão JSP que mostra a hora e data é:

```
Current time: <%= new java.util.Date() %>
```

Para simplificar essas expressões, existem algumas variáveis pré-definidas que você pode usar. As mais usadas são (Hall e Brown, 2003):

request, o objeto HttpServletRequest;
response, o objeto HttpServletResponse;
session, o objeto HttpSession (para manter informações de sessões[9])

out, um objeto PrintWriter usado como saída, ou seja, para exibir na página informações a serem enviadas ao usuário.

Por exemplo, para saber o nome da máquina cliente, basta inserir a seguinte expressão no seu JSP:

```
Your hostname: <%= request.getRemoteHost() %>
```

10.1.2 Scriplets JSP

Como você pôde observar na sessão anterior, usamos expressões quando queremos apenas exibir ao usuário o resultado de uma variável (pré-definida ou não) ou o retorno de um método. Se quisermos inserir código mais complexo em nossa página JSP, devemos usar os *scriplets*. Os

[9] *Uma sessão se refere a todas as requisições que um mesmo cliente faz a um servidor. No caso de servlets ou JSP pode ser criado um objeto HttpSession para manter o estado da seção . Este tema não é abordado neste livro. Para saber mais sobre o assunto, leia (Hall and Brown 2003; Perry 2004).*

scriplets permitem que você insira código no método do servlet (servlet resultante da tradução do JSP) que será construído para gerar a página web. Eles seguem a forma:

```
<% Código Java %>
```

Os *scriplets* possuem acesso as mesmas variáveis pré-definidas que as expressões. Assim, por exemplo, para exibir alguma informação ao usuário, você precisa usar a variável pré-definida *out*, como no exemplo abaixo retirado de (Hall, 1999):

```
<%
String queryData = request.getQueryString();
out.println("Attached GET data: " + queryData);
%>
```

10.1.3 Declarações

As declarações permitem a você definir métodos e variáveis que serão inseridos no corpo principal da classe (ou seja, fora do método que processa a requisição: *service*). Uma declaração possui a seguinte forma:

```
<%! Código Java %>
```

Como as declarações não geram nenhuma saída, geralmente elas são usadas em conjunto com expressões ou *scriplets* JSP. O exemplo a seguir, de (Hall, 1999), exibe o número de vezes que um JSP é requisitado desde a última reiniciação do servidor:

```
<%! private int accessCount = 0; %>
```

Accesses to page since server reboot:

```
<%= ++accessCount %>
```

11 Endereços Úteis na Internet

http://java.sun.com - Kit para desenvolvimento de aplicativos Java montado pelos criadores da linguagem. Sob este endereço você pode obter o compilador e outras ferramentas de desenvolvimento de aplicações Java para a sua plataforma de programação.

http://java.sun.com/docs/books/tutorial/index.html - Tutorial da linguagem Java, implementado pela Sun.

http://java.sun.com/docs/books/vmspec/index.html - Informações sobre a Java Virtual Machine.

http://java.sun.com/docs/index.html – Links para documentações da Plataforma Java.

http://java.sun.com/products/jdbc/index.html - Informações sobre JDBC.

http://industry.java.sun.com/products/jdbc/drivers - Lista de drivers JDBC disponíveis.

http://www.gamelan.com - Contem vários aplicativos Java e recursos para programadores.

http://www.javaworld.com - Revista eletrônica da linguagem java.

http://www.javasoft.com/applets - Links para vários *applets*, divididos por categorias: com jogos, animação, etc.

12 REFERÊNCIA BIBLIOGRÁFICA

(Albuquerque, 1999) Albuquerque, Fernando. **Programação Orientada a Objetos usando Java e UML.** Brasília: MSD, 1999.

(Booch, 1994) Booch, Grady. **Object Oriented Design with Applications.** Redwood City: The Benjamin/Cummings Publishing Company, 1994.

(Campione, 2000) Campione Mary, Walrath Kathy. **The Java Tutorial, Object-Oriented Programming for the Internet.** http://www.aw.com./cp/javaseries.html.

(Chan et al., 1999) Chan, Mark; Griffith, Steven & Iasi, Anthony. **Java 1001 Dicas de Programação.** São Paulo: Makron Books, 1999.

(Coad & Yourdon, 1993) Coad, Peter & Yourdon, Edward. **Projeto Baseado em Objetos.** Rio de Janeiro: Campus, 1993.

(Deitel, 2001) Deitel, Harvey M. **Java como Programar**. Porto Alegre: Bookman, 2001.

(Flanagan, 1999) Flanagan, David. **Java in a Nutshell – A Desktop Quick Reference** (Java 2SDK 1.2 e 1.3). 3rd Edition. Sebastopol: O´Reilly, 1999.

(Hamilton etal., 1997) Hamilton, Graham; Cattel, Rick & Fischer, Maydene. JDBC Database Access with Java. Massachusetts: Addison Wesley, 1997.

(Harold, 2000) Harold, Elliotte Rusty. **Brewing Java: A Tutorial.** http://sunsite.unc.edu/javafaq/javatutorial.html.

(Harold, 1997) Harold, Elliotte Rusty. **Java Network Programming.** Sebastopol: O'Reilly, 1997.

(Horstman & Cornell, 2000a) Horstmann, Cay & Cornell, Gary. **Core Java 2 - Volume 1 – Fundamentos**. Makron Books.

(Horstman & Cornell, 2000b) Horstmann, Cay & Cornell, Gary. **Core Java 2 - Volume 2 – Avançados**. Makron Books.

(Hunter & Crawford, 1998) Hunter, Jason & Crawford, Willian. **Java Servlet Programming.** Sebastopol: O'Reilly, 1998. http://www.ora.com/catalog/ jservlet/.

(Koffman & Wolz, 1999) Koffman, Elliot & Wolz, Ursula. **Problems Solving with Java.** USA: Addison-Wesley, 1999.

(Koosis & Koosis, 1999) Koosis, Donald & Koosis, David. **Programação com Java – Série para Dummies**. Rio de Janeiro: Campus, 1999.

(Lalani et al., 1997) Lalani, Suleiman San; Jamsa, Kris. **Java Biblioteca do Programador**. São Paulo: Makron Books, 1997.

(Lemay & Perkins, 1996) Lemay, Laura & Perkins Charles L. **Teach Yourself JAVA in 21 days.** Samsnet, 1996.

(Morgan , 1998) Morgan, Mike. **Using Java 1.2.** Indianapolis: QUE, 1998.

(Naughton, 1996) Naughton, Patrick. **Dominando o Java.** São Paulo: Makron Books, 1996.

(Oaks & Wong, 1997) Oaks, Scott & Wong, Henry. **Java Threads.** Cambridge: O'Reilly, 1997.

(Reese, 2000) Reese, George. **Database Programming with JDBC and Java**. http://www.ora.com/catalog/javadata/

(Ritchey, 1996) Ritchey, Tim. **Programando com Java! Beta 2.0.** Rio de Janeiro: Campus, 1996.

(Santos, 2003) Santos, Rafael. **Introdução à Programação Orientada a Objetos Usando Java**. São Paulo: SBC, Campus, 2003.