SÃOJUDAS UNIVERSIDADE	FTCE Faculdade de Tecnologia e Ciências Exatas

USJT - 2017 - Práticas de Programação

Professores	Teoria: Bonato, Hamilton Laboratório: Calvetti, Hamilton, Keity e Rodrigo

Aula: 11
Assunto: Leitura e gravação de arquivos

Conceitos básicos

Uma forma comum de persistência de dados não estruturados são os arquivos. Podemos salvar logs do sistema em arquivos texto. Podemos serializar objetos em arquivos, salvar imagens em arquivos, etc. Chamamos destas operações de I/O (pronuncia-se aiô), ou seja, input/output (entrada/saída) de dados. Todas as classes utilizadas no I/O de arquivos se encontram no pacote **`java.io`**.

A classe `File`

A classe básica para manipulação de arquivos em Java é a classe `File`. Ela representa um arquivo ou diretório (pastinha do Windows) como objeto no sistema. Ela não manipula arquivos diretamente, isto é, você não irá usá-la para ler ou gravar dados no arquivo. Porém ela pode dizer se o arquivo existe ou não e, existindo, qual é o seu tamanho, seu caminho (*path*) no disco, sua última data de modificação, etc.

Construtores

Construtor que encontra o arquivo no caminho atual
```
File arquivo = new File("arquivo.log");
```

Construtor que encontra o arquivo em qualquer caminho
```
File arquivo = new File("C:\\notas\\logs","arquivo.log");
```

Note que o uso da barra invertida é feito no Windows. E, como a barra invertida indica um caractere especial, como o "\n" ou o "\t", é necessário usar duas barras para que o "\n" de "\notas" não vire um pula linha.
Se o arquivo fosse ser instanciado no Unix, no Mac ou no Linux, ficaria assim:

```
File arquivo = new File("C:/notas/logs","arquivo.log");
```

Para tornar o seu código independente de plataforma, em vez da barra use `File.pathSeparator`.

```
File arquivo = new File("C: "+File.pathSeparator+ "notas" +
   File.pathSeparator + "logs","arquivo.log");
```

Alguns métodos

Testar a existência do arquivo ou diretório

```
   if(arquivo.exists()){
      //fazer coisas com o arquivo
   }
```

Testar se é arquivo ou diretório

```
   if(arquivo.isFile()){
      //fazer coisas com o arquivo
   } else if(arquivo.isDirectory()){
      //fazer coisa com o diretorio
   }
```

Obter os nomes dos arquivos de um diretório

```
   String[] arquivos = arquivo.list();
```

Conseguir a data de última alteração

```
      long tempo = arquivo.lastModified();
```

Conseguir o tamanho do arquivo

```
long tamanho = arquivo.length();
```

Há muitos outros métodos na API da classe File. Dê uma olhada no Javadoc.

I/O de baixo nível

É chamado de baixo nível pois lidam diretamente com dados binários. O arquivo resultante também é binário.

Uma vez que o arquivo está associado a um objeto File, podemos começara acessá-lo. Uma maneira básica de fazer isso é por meio de um ***stream*** (fluxo) de bytes. Há dois tipos de ***streams***: ***input stream*** para a leitura de dados e ***output stream*** para gravação.

FileOutputStream

O exemplo abaixo escreve em um conjunto de bytes em um arquivo.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

public class TestOutputStream {
   public static void main(String[] args) throws IOException {
      // set up file and stream
      File outFile = new File("sample1.data");
      FileOutputStream outStream = new FileOutputStream(outFile);
```

```
      // data to output
      byte[] byteArray = { 10, 20, 30, 40, 50, 60, 70, 80 }; // write data to
                                                             // the stream
      outStream.write(byteArray);
      // output done, so close the stream
      outStream.close();
   }
}
```

Sempre feche os arquivos para garantir que todo os dados tenham sido gravados.

O contrutor utilizado faz com que o conteúdo do arquivo seja totalmente substituído pelo conteúdo que está sendo gravado. Caso deseje acrescentar dados ao arquivo, utilize o construtor abaixo.

```
FileOutputStream(File file, boolean append)
```

Neste caso, se append for true, os dados serão acrescentados no final.

FileInputStream

O exemplo abaixo lê um arquivo byte a byte para um array de bytes e depois escreve o resultado na tela.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class TestInputStream {
   public static void main(String[] args) throws IOException {
      // set up file and stream
      File inFile = new File("sample1.data");
      FileInputStream inStream = new FileInputStream(inFile);
      // set up an array to read data in
      int fileSize = (int) inFile.length();
      byte[] byteArray = new byte[fileSize];
      // read data in and display them
      inStream.read(byteArray);
      for (int i = 0; i < fileSize; i++) {
         System.out.println(byteArray[i]);
      }
      // input done, so close the stream
      inStream.close();
   }
}
```

I/O de Alto Nível

Você pode usar um DataOutputStream para salvar tipos primitivos em um arquivo. Há nele métodos para gravar cada tipo. Os nomes dos métodos são autoexplicativos.

DataOutputStream

```java
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

public class TestDataOutputStream {
   public static void main(String[] args) throws IOException {
      // set up the streams
      File outFile = new File("sample2.data");
      FileOutputStream outFileStream = new FileOutputStream(outFile);
      DataOutputStream outDataStream = new DataOutputStream(outFileStream);
      // write values of primitive data types to the stream
      outDataStream.writeInt(987654321);
      outDataStream.writeLong(11111111L);
      outDataStream.writeFloat(22222222F);
      outDataStream.writeDouble(3333333D);
      outDataStream.writeChar('A');
      outDataStream.writeBoolean(true);
      // output done, so close the stream
      outDataStream.close();
   }
}
```

Note a ordem de instanciação: o DataOutputStream recebe um FileOutputStream que recebe um File. Na hora de gravar, os primitivos passam pelo DataOutputStream, são convertidos para bytes pelo FileOutputStream e são gravados no arquivo referenciado por File.

DataInputStream

Para ler os dados de volta, use um DataInputStream.

```java
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class TestDataInputStream {
   public static void main(String[] args) throws IOException {
      // set up file and stream
      File inFile = new File("sample2.data");
      FileInputStream inFileStream = new FileInputStream(inFile);
      DataInputStream inDataStream = new DataInputStream(inFileStream);
      // read values back from the stream and display them
      System.out.println(inDataStream.readInt());
      System.out.println(inDataStream.readLong());
      System.out.println(inDataStream.readFloat());
      System.out.println(inDataStream.readDouble());
      System.out.println(inDataStream.readChar());
      System.out.println(inDataStream.readBoolean());
      // input done, so close the stream
      inDataStream.close();
   }
}
```

Embora lidem com tipos primitivos de mais alto nível, o arquivo resultante é binário.

PrintWriter

O PrintWriter serve para escrever arquivos texto. Tem somente dois métodos de escrita, o print() e o println(), velhos conhecidos seus, que podem receber qualquer tipo primitivo como argumento.

O construtor precisa receber um FileOutputStream como argumento.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;

public class TestPrintWriter {
   public static void main(String[] args) throws IOException {
      // set up file and stream
      File outFile = new File("sample3.data");
      FileOutputStream outFileStream = new FileOutputStream(outFile);
      PrintWriter outStream = new PrintWriter(outFileStream);
      // write values of primitive data types to the stream
      outStream.println(987654321);
      outStream.println(11111111L);
      outStream.println(22222222F);
      outStream.println(33333333D);
      outStream.println('A');
      outStream.println(true);
      // output done, so close the stream
      outStream.close();
   }
}
```

Lembre-se que para acrescentar linhas no arquivo você deve usar o construtor `FileOutputStream(File file, boolean append)`.

BufferedReader

Para ler de um arquivo texto use um BufferedReader e um FileReader. O BufferedReader tem um método para ler o arquivo, o readLine, que sempre retorna uma String com a linha do arquivo lida.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class TestBufferedReader {
   public static void main(String[] args) throws IOException {
      // set up file and stream
      File inFile = new File("sample3.data");
      FileReader fileReader = new FileReader(inFile);
      BufferedReader bufReader = new BufferedReader(fileReader);
      String str;
```

```
      // get integer
      str = bufReader.readLine();
      int i = Integer.parseInt(str);
      // get long
      str = bufReader.readLine();
      long l = Long.parseLong(str);
      // get float
      str = bufReader.readLine();
      float f = Float.parseFloat(str);
      // get double
      str = bufReader.readLine();
      double d = Double.parseDouble(str);
      // get char
      str = bufReader.readLine();
      char c = str.charAt(0);
      // get boolean
      str = bufReader.readLine();
      Boolean boolObj = new Boolean(str);
      boolean b = boolObj.booleanValue();
      System.out.println(i);
      System.out.println(l);
      System.out.println(f);
      System.out.println(d);
      System.out.println(c);
      System.out.println(b);
      // input done, so close the stream
      bufReader.close();
   }
}
```

Scanner

É um outro modo de ler um arquivo texto. Você instancia o Scanner com um File e depois usa os métodos next, nextInt, nextDouble, etc. do scanner para lê-lo.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;

public class TestScanner {
   public static void main(String args[]) throws FileNotFoundException,
         IOException {
      // open the Scanner
      Scanner scanner = new Scanner(new File("sample3.data"));
      // get integer
      int i = scanner.nextInt();
      // get integer
      long l = scanner.nextLong();
      // get float
      float f = scanner.nextFloat();
      // get double
      double d = scanner.nextDouble();
      // get char
      char c = scanner.next().charAt(0);
      // get boolean
      boolean b = scanner.nextBoolean();
```

```
        System.out.println(i);
        System.out.println(l);
        System.out.println(f);
        System.out.println(d);
        System.out.println(c);
        System.out.println(b);
        // input done, so close the scanner
        scanner.close();
    }
}
```

Serialização de Objetos

Serializar objetos significa persistir o seu estado no disco, isto é, gravar o valor de suas variáveis de instância naquele momento.
Para que um objeto seja serializado pelo Tomcat basta fazer com que ele implemente a interface `java.io.Serializable`. Entretanto você pode querer fazer isso manualmente. Para isso você deve usar as classes ObjectOutputStream (para gravar) e ObjectInputStream (para ler) objetos serializados.

Tornando uma classe serializável

```
import java.io.Serializable;

public class Person implements Serializable {
   private String name;
   private int age;
   private char gender;

   public Person(String name, int age, char gender) {
      this.name = name;
      this.age = age;
      this.gender = gender;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public int getAge() {
      return age;
   }

   public void setAge(int age) {
      this.age = age;
   }

   public char getGender() {
      return gender;
   }
```

```
   public void setGender(char gender) {
      this.gender = gender;
   }
}
```

ObjectOutputStream

Você pode salvar um ou mais objetos, mesmo que sejam diferentes, em um mesmo arquivo. O exemplo abaixo salva 10 objetos do Person.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class TestObjectOutputStream {
   public static void main(String[] args) throws IOException {
      // set up the streams
      File outFile = new File("objects.dat");
      FileOutputStream outFileStream = new FileOutputStream(outFile);
      ObjectOutputStream outObjectStream = new ObjectOutputStream(
            outFileStream);
      // write serializable Person objects one at a time
      Person person;
      for (int i = 0; i < 10; i++) {
         person = new Person("Mr. Espresso" + i, 20 + i, 'M');
         outObjectStream.writeObject(person);
      }
      // output done, so close the stream
      outObjectStream.close();
   }
}
```

ObjectInputStream

Serve para ler os objetos serializados e instanciar objetos com eles. Caso tenha serializado objetos de tipos diferentes, é necessário ter cuidado na leitura e instanciação para não gerar um erro de conversão - `ClassCastException`.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class TestObjectInputStream {
   public static void main(String[] args) throws ClassNotFoundException,
         IOException {
      // set up file and stream
      File inFile = new File("objects.dat");
      FileInputStream inFileStream = new FileInputStream(inFile);
      ObjectInputStream inObjectStream = new ObjectInputStream(inFileStream);
      // read the Person objects from a file
      Person person;
      for (int i = 0; i < 10; i++) {
         person = (Person) inObjectStream.readObject();
         System.out.println(person.getName() + "      " + person.getAge()
```

```
                + "      " + person.getGender());
        }
        // input done, so close the stream
        inObjectStream.close();
    }
}
```

Encontrando caminhos com Servlets

Na aula passada vimos dois métodos para retornar caminhos a partir da URL:

`String getRequestURI()`: retorna a URL que chamou o servlet a partir do nome do servidor. Por exemplo, se a URL foi **http://localhost:8080/projeto/controller.do**, o retorno será **/projeto/controller.do**.

`String getContextPath()`: retorna o contexto da URL retornada por getRequestURI. Usando o mesmo exemplo acima, este método retornaria **/projeto**. Note que ele sempre irá começar com uma barra e terminar sem a barra. Se a requisição vier do contexto raiz (/), o retorno será uma string vazia "".

Porém, estes dois métodos não nos serão úteis para a gravação de arquivos no servidor, pois o que precisamos é do caminho absoluto do arquivo no disco. Para isso iremos usar:

- a classe `FilterConfig`, que é usada pelo container de servlet para passar informações para o filtro.

- desta classe usaremos o método `getServletContext`() para pegar o contexto do servlet.

- do ServletContext usamos o método `getRealPath(File.separator)` para retornar o caminho absoluto do arquivo no disco. Lembre-se que o File.separator é para indicar o tipo de barra - normal para Linux, invertida para Windows - que iremos utilizar.

Veja o exemplo abaixo:

LogFilter alterado para gravar o Log. Para funcionar crie uma pasta log no seu Webcontent. Mas atenção. O arquivo gerado não ira aparecer no Eclipse, mas nos diretórios do Tomcat. Para encontrá-lo, vá até o caminho que é impresso na console do Eclipse a cada gravação.

synchronized: os métodos de gravação em arquivo não são thread safe. Lembre-se que seu sistema é multiusuário e que váias pessoas irão gravar concorrentemente no aquivo de log. A palavra chave synchronized enfileira as requisições na entrada do bloco demarcado por ela e só deixa uma thread passar por vez.

```
package filter;
```

```
import java.io.File;
import java.io.IOException;
import java.util.Calendar;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

import model.Usuario;
import utils.Log;

@WebFilter("/*")
public class LogFilter implements Filter {

   FilterConfig filterConfig = null;

   public void destroy() {

   }

   public void doFilter(ServletRequest request, ServletResponse response,
         FilterChain chain) throws IOException, ServletException {
      // place your code here
      request.setCharacterEncoding("UTF-8");
      response.setCharacterEncoding("UTF-8");
      HttpServletRequest req = (HttpServletRequest) request;
      HttpSession session = req.getSession();
      Usuario usuario = (Usuario) session.getAttribute("logado");

      String comando = req.getParameter("command");
      if (comando == null) {
         comando = req.getRequestURI();
      }
      Calendar timestamp = Calendar.getInstance();
      String textoLog = "";
      ServletContext servletContext = filterConfig.getServletContext();
      String contextPath = servletContext.getRealPath(File.separator);

      if (usuario == null) {
         textoLog = String
               .format("[%1$tA, %1$tB %1$td, %1$tY %1$tZ
%1$tI:%1$tM:%1$tS:%1$tL %tp] %s\n",
                     timestamp, comando);
      } else {
         textoLog = String
               .format("[%1$tA, %1$tB %1$td, %1$tY %1$tZ
%1$tI:%1$tM:%1$tS:%1$tL %tp] %s -> %s\n",
                     timestamp, usuario.getUsername(), comando);
      }
      synchronized (textoLog) {
```

```
        Log arqLog = new Log();
        //arqLog.abrir(Log.NOME);
        arqLog.abrir(contextPath + File.separator + "log" + File.separator
              + Log.NOME);
        arqLog.escrever(textoLog);
        arqLog.fechar();
     }
     // pass the request along the filter chain
     chain.doFilter(request, response);
  }

  public void init(FilterConfig fConfig) throws ServletException {
     this.filterConfig = fConfig;
  }

}
```

Log: usa um PrintWriter para escrever no aquivo texto de logs.

```
package utils;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;

public class Log {
   public static final String NOME = "rastreamento.txt";
   private PrintWriter arquivo;

   public void abrir(String nome) throws IOException {
      File outFile = new File(nome);
      FileOutputStream outFileStream;

      System.out.println("Procure o arquivo em " + nome);
      if (outFile.exists()) {
         outFileStream = new FileOutputStream(outFile, true);
      } else {
         outFileStream = new FileOutputStream(outFile);
      }
      arquivo = new PrintWriter(outFileStream);
   }

   public void escrever(String texto) throws IOException {
      arquivo.println(texto);
      arquivo.flush();
   }

   public void fechar() throws IOException {
      arquivo.close();
   }

}
```

Exercícios

1. Revisão da Teoria
a. Importe o arquivo exemplo_clientes.war. Faça deployment no Tomcat e rode. Depois analise o código para entendê-lo.

2. Exercício para Nota (correção em aula)

Implemente a partir do código com o Filtro do projeto de Países da aula passada e comece a armazenar os logs do filtro de log em um arquivo texto.

Bibliografia

WU, C. Thomas; **An Introduction to Object Oriented Programming with Java**; McGraw Hill; 2010; New York. 1009 p. ISBN 978–0–07–352330–9;

BASHAM, Bryan; SIERRA, Kathy; BATES, Bert. **Use a cabeça!: Servlets & JSP**. 2. ed. Rio de Janeiro: Alta Books, 2008-2010. xxxii, 879 p. ISBN 9788576082941 (broch.)