Unidade 2 – Parte 2

JAVA I/O

O sistema de entrada e saída, tratado carinhosamente pelos desenvolvedores como sistema de I/O, da linguagem Java é extremamente rico. O pacote java.io fornece uma série de classes para tratamento de fluxo de dados, serialização e sistema de arquivos. Como o pacote java.io é muito extenso estaremos focando nas principais classes e interfaces, mostrando exemplos mais comuns.

A classe File

A classe File fornece uma série de métodos para tratamento de arquivos e diretórios. Para exibir a lista de arquivos de um diretório poderemos utilizar o método *list* da classe.

```
import java.io.*;

public class Arq
{
	public static void main (String [] args)
	{
		try
		{
			File path = new File("."); // "c:/"

			String [] list = path.list();

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

		}
		catch(Exception e)
		{
```

```
                System.out.println(e);
            }
        }
}
```

Em algumas situações desejamos listar apenas alguns tipos de arquivos. Neste caso deveremos criar um filtro de arquivos e utilizar este filtro em conjunto com o método list. Filtros são classes que deverão implementar a interface *FilenameFilter*. Esta interface exige a implementação do método *accept* que deverá retornar um boolean indicando se deve ou não considerar o arquivo.

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

public class ArqFiltro
{
    public static void main (String [] args)
    {
        try
        {
            GenericFilter filtro = new GenericFilter();

            filtro.add("java");
            filtro.add("doc");


            File path = new File(".");

            String [] list = path.list( filtro );

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

        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }

}
```

```
class GenericFilter implements FilenameFilter
{
	private Vector extensoes;

	public GenericFilter()
	{
		extensoes = new Vector();
	}

	public void add (String extensao)
	{
		extensoes.addElement(extensao);
	}

	public boolean accept (File dir, String name)
	{
		String f = (new File(name)).getName();
		int pos_ponto = f.indexOf(".");
		if( pos_ponto == -1 )
		{
			return false;
		}
		else
		{
			for( int i=0; i<extensoes.size(); ++i)
			{
				String s = f.substring(pos_ponto+1,f.length());
				if( s.equalsIgnoreCase((String) extensoes.get(i)) )
				{
					return true;
				}
			}
			return false;
		}
	}

}
```

A classe File pode ser usada para obter informações de arquivos e diretórios:

```
import java.io.*;

public class InfoFile
{
	public static void main (String [] args)
	{
		try
		{
			File file = new File("index.html");

			System.out.println( file.getAbsolutePath() );
			System.out.println( file.canRead() );
			System.out.println( file.canWrite() );
			System.out.println( file.getName() );
			System.out.println( file.getParent() );
			System.out.println( file.length() );
			System.out.println( file.lastModified() );
			System.out.println( file.isFile() );
			System.out.println( file.isDirectory() );

		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}

}
```

Stream

Toda a biblioteca de classes de I/O está baseada no conceito de Stream. Um stream é uma fonte de dados que pode ser utilizada para leitura e gravação. Os streams se dividem basicamente em dois grupos: streams de leitura e streams de escrita. As classes que participam da estrutura de *InputStream* ou *Reader* são classes para leituras, e as classes que participam da estrutura de *OutputStream* ou *Writer* são classes utilizadas para escrita.

Uma vez que existem diversos tipos de dados, existem conseqüentemente classes para tratar estes tipos de dados adequadamente. As classes para operações de leitura são as seguintes: *ByteArrayInputStream, StringBufferInputStream, FileInputStream, PipedInputStream, SequenceInputStream e FilterInputStream*. Da mesma forma, existem diversar classes para tratar a escrita adequadamente: *ByteArrayOutputStream, FileOutputStream, PipedOutputStream, FilterOutputStream*.

Para exemplificar o uso de arquivos textos, vamos utilizar a classe *FileOutputStream* e *PrintStream*:

```
import java.io.*;

public class GravaTxt
{
	public static void main (String [] args)
	{
		try
		{
			String nome_arq = "teste.txt";   // "/home/heleno/teste.txt"
			FileOutputStream file = new FileOutputStream( nome_arq );
			PrintStream out = new PrintStream( file );

			out.println( "Primeira linha." );
			out.println( "Segunda linha." );

		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
}
```

Como exemplo de leitura de arquivos texto utilizaremos as classes BufferedReader e InputStreamReader:

```
import java.io.*;

public class LeTxt
{
	public static void main (String [] args)
	{
		try
		{
			String nome_arq = "teste.txt";
			FileInputStream file = new FileInputStream( nome_arq );
			InputStreamReader isr = new InputStreamReader( file );
			BufferedReader in = new BufferedReader( isr );

			String linha = in.readLine();
			while( linha != null )
			{
				System.out.println( linha );
				linha = in.readLine();
			}
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
}
```

Este exemplo simples mostra claramente a relação existente entre diversas classes do pacote java.io. Precisamos de um objeto da classe *BufferedReader*, no entanto o construtor dela necessita de um objeto *Reader*. Utilizamos então um *InputStreamReader* que é uma subclasse de *Reader*, no entanto esta precisa de um *InputStream* como parâmetro do seu construtor. Utilizamos um *FileInputStream* que é uma subclasse de *InputStream*, e finalmente o construtor de *FileInputStream* está baseado na string com o nome do stream.

No entanto poderíamos utilizar um *FileReader*, o que tornaria menos complexo o nosso exemplo, para tal substituiríamos a criação do objeto in da seguinte forma:

```
BufferedReader in = new BufferedReader( new FileReader(nome_arq) );
```

A classe *BufferedReader* obtém dados de um stream de entrada, criando um “buffer” de caracteres, otimizando o processo de leitura.

Gravando Objetos

Quando estamos lidando com programas orientados a objeto, mais cedo ou mais tarde será necessário gravar objetos em streams, esta necessidade é tão grande que o Java criou um mecanismo chamado de serialização de objetos. A serialização de objetos consiste em transformá-los em uma seqüência de bytes e gravá-los de maneira que na hora de restaurá-los seja possível recriar as instâncias e suas referências. Para que um objeto de uma classe possa ser “serializado”, esta classe deverá implementar a interface *Serializable*.

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

public class GravandoObj
{
	public static void main (String [] args)
	{
		try
		{
			Vector lista_g = new Vector();
			lista_g.addElement( new Pessoa("Pedro") );
			lista_g.addElement( new Pessoa("Joao") );

			FileOutputStream file_g = new FileOutputStream( "pessoas.dat" );
			ObjectOutputStream out = new ObjectOutputStream( file_g );

			out.writeObject( lista_g );
			out.close();



			// Lendo Objts
			Vector lista_l = null;

			FileInputStream file_l = new FileInputStream( "pessoas.dat" );
			ObjectInputStream in = new ObjectInputStream( file_l );
```

```
                    lista_l = (Vector) in.readObject();
                    in.close();



                    Iterator it = lista_l.iterator();
                    while( it.hasNext() )
                    {
                        Pessoa p = (Pessoa) it.next();
                        System.out.println( p.getNome() );
                    }


                }
                catch(Exception e)
                {
                    System.out.println(e);
                }
        }
}


class Pessoa implements Serializable
{
        private String nome;

        Pessoa(String nome)
        {
            this.nome = nome;
        }

        public void setNome(String nome)
        {
            this.nome = nome;
        }

        public String getNome()
        {
            return nome;
        }
}
```