UNIVERSIDADE DO ESTADO DE SANTA CATARINA
DCC – Departamento de Ciência da Computação
Joinville – SC

CURSO: Bacharelado em Ciência da Computação
DISCIPLINA: POO0001 – Programação Orientada a Objetos
PROFESSOR: Diego Buchinger
AULA 03 – Introdução à Java – Parte II

1. *WRAPPERS*

Agora que já é de seu entendimento o conceito de CLASSES e OBJETOS, podemos nos aprofundar em um detalhe de tipagem da linguagem de programação Java. Para o paradigma orientado a objetos, idealmente tudo deveria ser tratado como CLASSE ou OBJETO. Contudo, é comum que as linguagens de programação ofereçam tipos primitivos para serem declarados e utilizados no programa. É algo nesse sentido que é feito no Java: você percebeu que as variáveis inteiras de ponto flutuante, por exemplo, não precisam ser instanciadas com o operador *new*? Isto ocorre porque elas são tratadas como tipos primitivos e podem ser atribuídas de forma direta.

Entretanto, para algumas situações pode ser interessante tratar tais tipos como objetos também. Para isto existem os *wrappers* que são definições de CLASSES para os tipos primitivos da linguagem. No Java, os *wrappers* possuem os seguintes nomes:

```
boolean → Boolean        short → Short        float → Float
char → Character         int → Integer        double → Double
byte → Byte              long → Long          void → Void
```

Como são CLASSES, a sua instanciação deve ocorrer usando um método construtor através do operador *new*, como qualquer outro OBJETO. Um exemplo de instanciação de um objeto inteiro é:

```
Integer meu_inteiro = new Integer( 65 );
```

Como são objetos, os *wrappers* possuem métodos que podem ser úteis para diversas situações. Para verificar quais métodos, atributos e constantes estão disponíveis para uso, podemos requisitar ao netbeans uma listagem auxiliar, digitando o nome da variável e pressionando ponto (.), como na Figura 1. Se a lista sumir é possível forçar o seu aparecimento com *Ctrl+Espaço* ou apagando e redigitando o caractere ponto. Faça um teste e veja quais métodos estão disponíveis para alguns tipos primitivos.

Os métodos dos OBJETOS *wrappers* contudo, não são usualmente tão interessantes ao compará-los com os métodos e constantes referente a definição de sua CLASSE. Lembre-se que para acessar os métodos da classe (ou seja, *static*) deve-se utilizar o nome da classe e o operador ponto. Talvez um dos métodos mais interessantes para nós neste momento, e que vai ser útil em diversas situações é o *parser*, que converte uma string no tipo do *wrapper*. Na verdade existem dois métodos que fazem esta tarefa: o *parseTipo(String)* [onde Tipo é o tipo primitivo do *wrapper*; por exemplo, parseInt] e o *valueOf(String)*. A diferença é que o primeiro retorna um tipo primitivo e o segundo retorna um objeto:

```
int a = Integer.parseInt( "65" );
Integer b = Integer.valueOf( "-1250" );
```

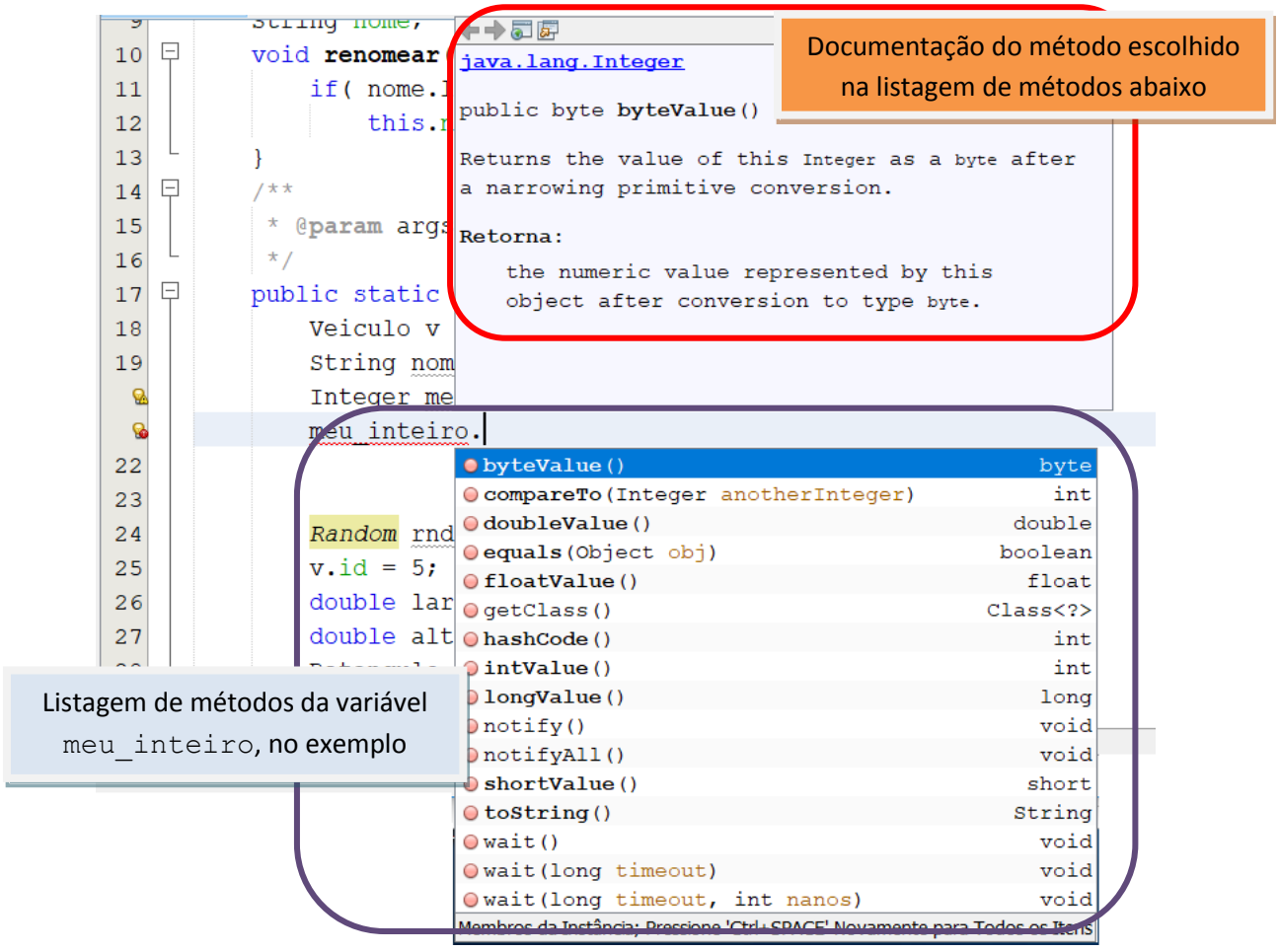


FIGURA 1. Listagem de constantes, atributos e métodos acessíveis pelo Netbeans.

Algumas CLASSES dos *wrappers* possuem também a definição de constantes que são identificadas, como boa prática de programação, por palavras todas em letras maiúsculas. Dois exemplos clássicos destas constantes são o MAX_VALUE e o MIN_VALUE que identificam, respectivamente, o maior e o menor valor que aquele determinado objeto pode representar.

Em Java, para representarmos uma constante, utilizamos a palavra chave ***final*** que deve aparecer antes da identificação do tipo da variável ou atributo e depois do seu escopo. O valor de uma constante deve ser definido na linha de sua declaração, ou então nos métodos construtores do objeto!

2. ARRAYS – VETORES E MATRIZES

A definição e uso de arrays em Java é bastante semelhante ao que ocorre na linguagem C, mas possui suas peculiaridades. Para o Java, os *arrays* são tratados como OBJETOS e devem ser instanciados como tal. Assim, a declaração e instanciação de um vetor (*array* unidimensional) pode ser feita da seguinte forma:

```
int[] vetor = new int[25];
int[] vet = new int[]{1,20,3,40};
```

Note que não utilizamos parênteses nesta instanciação e que o tamanho do vetor só é definido na atribuição. Inclusive, tal tamanho pode ser fixo, como no exemplo, mas também pode receber

um valor provindo de um atributo ou de uma variável. No exemplo apresentado foi utilizado o tipo primitivo inteiro, mas qualquer tipo primitivo ou composto pode ser utilizado na declaração e instanciação de vetores, com exceção do tipo *void*.

A instanciação de *arrays* multidimensionais (matrizes ou outros) merece uma atenção especial pois pode ser feita de duas formas distintas. A primeira forma de instanciação é semelhante ao modo de instanciar um vetor, definindo-se um valor para todas as dimensões do *array*. Outra forma de instanciação é definir apenas parte das dimensões (sempre da dimensão mais da esquerda para a direita) e posteriormente definir as demais dimensões de forma isolada.

```
int[][] matriz = new int[10][7];
int[][] mat = new int[10][];
for(int i=0; i<10; i++)
    mat[i] = new int[ i+1 ];
```

Na primeira forma de instanciação teremos um *array* multidimensional no estilo matriz 10x7, com 10 linhas e cada linha contendo 7 colunas. Já na segunda forma de instanciação do exemplo, teremos um *array* multidimensional com 10 linhas, no qual a primeira linha (linha 0) terá apenas uma coluna, a segunda linha (linha 1) terá apenas duas colunas, e assim sucessivamente até a décima linha (linha 9) que terá 10 colunas.

> **Atenção:** no Java, uma sequência de caracteres isolados por aspas duplas não é tratada como um vetor de caracteres mas sim como uma objeto String!
>
> Dessa forma, **NÃO** é possível declarar um vetor de caracteres no estilo C:
>
> ```
> char[] nome = "teste";
> ```

2.1. Passagem de *Arrays* como Parâmetros de uma Função

A linguagem Java permite a passagem de *arrays* como parâmetros de uma função por referência de maneira simplificada. Entretanto, é importante entender como esta passagem funciona, tendo um cuidado extra ao utilizá-lo dentro da função. Como já mencionado, um *array* é passado sempre por referência para uma função. Assim, caso um valor do *array* seja alterado dentro da função, ele será alterado para o objeto original também. Entretanto, se um novo *array* for instanciado para o argumento da função, ele deixará de referenciar o *array* original, deixando-o intacto e passará a apontar/referenciar este novo objeto *array* criado. Replique o seguinte código fonte de exemplo e veja o resultado de sua execução.

> **ATENÇÃO**: essa propriedade da passagem de *arrays* como parâmetros, na verdade se aplica a todo e qualquer OBJETO. Sempre que um OBJETO (tipo composto) for passado como parâmetro para uma função, sua passagem é feita por referência e toda alteração realizada implicará também no objeto original. Entretanto, se uma nova instância for criada para aquele argumento, perde-se a referência do objeto original.
>
> Já os tipos primitivos são sempre passados por valor – uma cópia de seu valor é atribuída ao argumento da função – não alterando de forma alguma o valor original. Caso seja de interesse passar um valor primitivo por referência, use os *wrappers*.

Arquivo: POO.java

```
 1.  public class POO{
        ...
13.     static void foo(float[] vA, float[] vB){
14.        vB = new float[5];
15.        for(int i=0; i < vA.length; i++){
16.            vA[i] = i + 1.5f;
17.            vB[i] = i + 1.5f;
18.        }
19.     }
20.     public static void main(String[] args) {
21.        float[] vA = new float[5];
22.        float[] vB = new float[5];
23.        foo( vA , vB );
24.        for(int i=0; i < vA.length; i++)
25.            System.out.println(i+" - vA:"+vA[i]+" / vB:"+vB[i]);
26.     }
27.  }
```

3. CLASSES STRING, MATH, ARRAYS E RANDOM

Duas classes extremamente úteis e que devem ser de conhecimento de qualquer desenvolvedor Java são as classes String e Math. Agora que você já conhece o conceito de CLASSES, OBJETOS, e a propriedade *static* será mais fácil entender como usá-las e o porquê da sintaxe de seu uso. Existem diversas outras classes implementadas na linguagem Java, inclusive algumas que implementam o comportamento das principais estruturas de dados da computação: filas, pilhas, listas, árvores, entre outras. Mas vamos nos ater as mais básicas neste momento.

3.1. String.java

A classe String implementa uma estrutura de dados que trabalha com, pasme, strings. É comum denominarmos os vetores de caracteres como string, mas como na linguagem Java temos uma estrutura/CLASSE pronta para utilizar strings, é usual chamarmos os vetores de caracteres apenas de vetores de caracteres, e os objetos da CLASSE String de strings. Você pode estar se perguntando: Ok, mas qual a diferença entre vetores de caracteres e a String? Na verdade, são muitas. O vetor de caracteres é apenas um vetor de um tipo primitivo de dado, com a limitação de tamanho típica de um vetor tradicional e as operações básicas sobre vetores como acessar e modificar um elemento do vetor e consultar o seu tamanho. Um objeto String, por outro lado, permite o redimensionamento do tamanho de maneira simples e oferece uma vasta coleção de operações (métodos) prontos para serem utilizados.

Para utilizar a classe String não é preciso realizar nenhuma importação, ela já está disponível para uso de forma direta.

Por se tratar de uma CLASSE, toda String deve ser instanciada como um objeto, contudo, existe uma sintaxe que já usamos diversas vezes que simplifica a instanciação de uma String: é o uso das aspas duplas. Toda texto que escrevemos no código com aspas duplas nada mais é do que um OBJETO da CLASSE String. Inclusive, pode-se utilizar os métodos disponíveis aos objetos String.

```
String nome = new String("Diego");
String nome = "Diego";
System.out.println( "tisti".replace('i', 'e') );
```

Falando nos métodos disponíveis aos OBJETOS String, há uma vasta lista de métodos implementados. A lista abaixo apresenta alguns métodos que podem ser úteis em algum momento:

- `charAt( int index )` – retorna o caractere do índice especificado;
- `contains( CharSequence s )` – verifica se o objeto contém uma substring s;
- `compareTo( String s )` – compara lexicograficamente o objeto com a String parâmetro. O retorno é um valor negativo se o objeto precede o argumento, o valor zero caso as duas Strings sejam iguais ou um valor positivo se o argumento precede o objeto;
- `endsWith( String suffix )` – verifica se o objeto contém o sufixo especificado;
- `length()` – retorna o tamanho da String;
- `replace( CharSequence a, CharSequence b )` – retorna um novo objeto String substituindo todas as ocorrências da substring *a* pela string *b*. OBS: para alterar a String original é necessário atribuir este resultado à String;
- `split( String separador )` – divide uma String em pedaços utilizando o separador passado como parâmetro. Retorna um array de String com cada pedaço;
- `toLowerCase()` – retorna um novo objeto String com as letras todas minúsculas;
- `toUpperCase()` – retorna um novo objeto String com as letras todas maiúsculas;

Uma última questão de muita importância é a questão de comparação entre objetos. Imagine a seguinte situação apresentada no trecho de código abaixo:

```
String s1 = new String("Mr. Striffe");
String s2 = new String("Mr. Striffe");
if( s1 == s2 ){ ... }
```

O resultado desta comparação será falso, pois o operador == compara se os dois objetos em questão são o mesmo (referenciam o mesmo conjunto de dados), o que não é o caso no exemplo. Para comparar se o valor de dois objetos é igual deve-se utilizar um método que faz esta comparação. No caso da classe String, existe o método compareTo() e compareToIgnoreCase() que comparam o valor de duas strings é exatamente igual, sendo que o primeiro método é *case sensitive* (considera letras minúsculas e maiúsculas como diferentes) e o segundo não.

ATENÇÃO: se por acaso você testar a seguinte comparação terá um resultado verdade como resposta, mas por quê?

```
String n1 = "Mr. Striffe";
String n2 = "Mr. Striffe";
if( n1==n2 )
```

Isto ocorre porque a maioria das linguagens de programação que possuem uma etapa de compilação, otimizam o código de alguma forma. Nesta situação, como as strings constantes são exatamente iguais, elas são otimizadas e criadas como um único objeto. Esta otimização, provavelmente seria algo assim:

```
String _otimiz = "Mr. Striffe";
String n1 = _otimiz;
String n2 = _otimiz;
if( n1==n2 )
```

3.2 Math.java

A classe Math implementa uma coleção de constantes (como o PI e o número de Euler E) e operações matemáticas clássicas. Não há necessidade realizar importação desta classe, nem de instanciar um objeto Math para utilizar esta classe, pois todos os métodos públicos da classe são estáticos. Na verdade, não há nem mesmo a possibilidade de instanciar um objeto. Observe o seguinte trecho retirado da classe Math:

```
/** Don't let anyone instantiate this class. */
private Math() {}
```

*OBS: o espaçamento foi adaptado

Existem diversos métodos matemáticos implementados na classe Math. A seguir são listados alguns dos métodos que possivelmente são mais utilizados:

- `abs( num )` – retorna o valor absoluto (sem sinal) de um valor int, long, float ou double;
- `acos( doubel ) / asin( double ) / atan( double )` – retorna o arco cosseno, arco seno ou arco tangente, respectivamente, de um dado ângulo (ambos o retorno e o parâmetro são valores em radianos);
- `cos( double ) / sin( double ) / tan(double )` – retorna o cosseno, o seno ou a tangente de um dado ângulo (ambos o retorno e o parâmetro são valores em radianos);
- `ceil( double )` – retorna o menor valor real que é maior ou igual ao valor passado por parâmetro, considerando os inteiros;
- `floor()` – retorna o maior valor real que é menor ou igual ao valor passado por parâmetro, considerando os inteiros;
- `log( double ) / log10( double )` – retorna o logaritmo natural (base *e*) ou o logaritmo na base 10 de um dado valor;
- `max( num, num ) / min( num, num )` – retorna o maior ou menor valor, respectivamente, entre os valores passados por parâmetro;
- `pow( double, double )` – retorna o resultado da exponenciação do primeiro parâmetro pelo segundo;
- `toDegree( double ) / toRadians( double )` – converte e retorna um valor em radianos para graus, ou um valor de graus para radianos, respectivamente.

3.3 Arrays.java

A CLASSE Arrays, como o nome já sugere, é uma CLASSE utilitária para operações usuais em arrays. Tal biblioteca pode ser importada com a instrução: `import java.util.Arrays;` Assim como na CLASSE Math, não é necessário, nem possível, instanciar um OBJETO desta CLASSE, sendo que todos os métodos públicos da CLASSE são estáticos. Existem diversos métodos matemáticos implementados na classe Math. Alguns métodos interessantes são:

- `sort( tipo[], int, int )` – ordena o vetor passado por parâmetro usando o comparador do tipo (ex: números em ordem crescente, strings em ordem lexicográfica). O segundo e terceiro parâmetros indicam a partir de qual índice e até qual índice, respectivamente, a ordenação deve ser realizada;

- `fill( tipo[], int, int, tipo )` – preenche o vetor passado por parâmetro com elementos especificados no quarto parâmetro. O segundo e terceiro parâmetros indicam a partir de qual índice e até qual índice, respectivamente, a ordenação deve ser realizada;
- `bool equals( tipo[], tipo[] )` – compara se dois arrays são iguais. Estes vetores são considerados iguais se possuem o mesmo tamanho e cada elemento é considerado igual.

3.4 Random.java

Uma quarta CLASSE que pode ser de bom uso para diversos tipos de aplicações é a Random.java que provê geradores de números pseudoaleatórios. Tal biblioteca pode ser importada com a instrução: `import java.util.Random;`

A classe Random não provê métodos estáticos, exigindo que um objeto seja instanciado para que seja possível fazer uso das funcionalidades de geração de valores aleatórios. Basicamente deve-se instanciar um objeto Random, opcionalmente definir uma semente de geração de números aleatórios (lembre-se que a sequência de números aleatórios será sempre a mesma para uma dada semente), e usar os métodos que retornam um valor pseudoaleatório. Os principais métodos são:

- `nextInt()` – gera um número aleatório inteiro qualquer;
- `nextInt( int )` – gera um número aleatório entre 0 e o argumento informado;
- `nextFloat() / nextDouble()` – gera um número aleatório entre 0 e 1;
- *`nextBytes( bytes[] )`* – preenche um vetor previamente instanciado de bytes com valores aleatórios para cada posição do vetor

```
Random rand = new Random();
double valA = rand.nextDouble();
int    valB = rand.nextInt( 5 );
byte[] vetor = new byte[25];
rand.nextBytes( vetor );
```

4. ARQUIVOS

Para finalizar o material desta aula, vamos estudar sobre como trabalhar com arquivos em Java. Agora que você já tem uma noção sobre os conceitos de CLASSE e OBJETOS, e também já sabe realizar conversões de String para valores numéricos, provavelmente será mais fácil trabalhar com arquivos. Por outro lado, ainda não tratamos do conceito de exceções (*exceptions*), portanto, apenas por enquanto, altere ligeiramente o seu método *main* para a seguinte forma:

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

Vamos conhecer então as principais classes nativas do Java para manipular arquivos.

4.1. Manipulando Arquivos

A primeira classe que vamos ver é a CLASSE *File*, que pode ser importada da seguinte forma:

```
import java.io.File;
```

A CLASSE *File* provê acesso ao sistema de arquivos do sistema operacional, permitindo a manipulação de arquivos e diretórios, mas por si só não oferece poder de edição do conteúdo dos

arquivos. A primeira coisa que geralmente deve ser feita é a criação de uma instância de um objeto *File* para que se possa trabalhar com ele posteriormente.

```
File caminho = new File("meu-arquivo.txt");
```

O caminho fornecido pode ser tanto local (considerando o diretório do projeto) como absoluto, e pode se tratar de um nome de arquivo ou de um diretório. Note que esta instanciação não cria efetivamente um arquivo ou um diretório. Para fazer isto, caso o arquivo ou diretório ainda não existam, podemos utilizar as expressões:

```
caminho.createNewFile();
caminho.mkdir();
```

Com um OBJETO *File* instanciado é possível realizar diversas outras operações. Para ver uma lista dos métodos, utilize o suporte do Netbeans que apresenta os métodos disponíveis (*Ctrl+Enter*). Apenas para citar alguns métodos disponíveis, é possível verificar se o arquivo ou diretório já existe [*exists()* / *isDirectory()*], qual o seu tamanho [*lenght()*], renomear [*renameTo()*] ou remover [*delete()*], ou ainda obter uma lista de arquivos dentro de um diretório [*list() / listFiles()*].

4.2. Escrevendo em Arquivos

Tendo uma instancia de um OBJETO *File* referente a um arquivo, podemos utilizar este OBJETO para realizar operações de escrita no arquivo com auxílio da CLASSE *FileOutputStream*. Também é possível escrever sem este OBJETO, fornecendo apenas o caminho do arquivo que se deseja editar. Para escrever em um arquivo utilizando esta CLASSE, deve-se criar um novo OBJETO desta CLASSE e utilizá-lo para escrever no arquivo. Ao final do processo de escrita é importante lembrar de fechar o arquivo. O exemplo a seguir mostra um exemplo de uso.

```
import java.io.FileOutputStream;
(...)
File arquivo = new File("meu-arquivo.txt");
FileOutputStream output = new FileOutputStream(arquivo, true);
String conteudo = "Teste de escrita";
float valor = 54.7f;
output.write( conteudo.getBytes() );
output.write( String.valueOf(valor).getBytes() );
output.close();
```

Forma enxuta de escrever:

```
String aux = String.valueOf(valor);
aux.getBytes();
```

Primeiramente perceba que para criar um OBJETO de FileOutputStream, foi utilizado um OBJETO do tipo arquivo e um valor booleano. O primeiro parâmetro informa qual o arquivo que deve ser usado para a escrita enquanto que o segundo parâmetro indica se a operação será do tipo *append*, ou seja, acrescentar mais texto sem apagar o que já existe, ou um sobrescrita, apagando o conteúdo já existente.

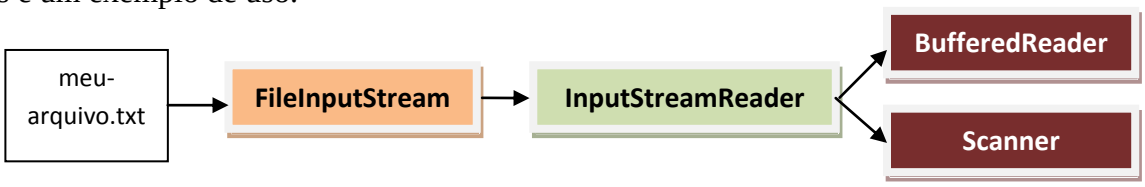
Note também que é possível escrever apenas bytes no arquivo (um byte na forma de inteiro ou um vetor de bytes), mas esta conversão pode ser realizada facilmente com o uso da CLASSE String que possui um método *getBytes()*, o qual retorna o conteúdo da string em uma representação na forma de vetor de bytes. Observe ainda no exemplo acima como um valor real foi escrito no

arquivo. Ele foi primeiramente convertido para String e então, na mesma linha, foi transformado em um vetor de bytes.

4.3. Lendo Conteúdo de Arquivos

De maneira semelhante à CLASSE anteriormente apresentada, com uma instancia de um OBJETO *File* referente a um arquivo, podemos utilizar este OBJETO para realizar operações de leitura no arquivo com auxílio da CLASSE *FileInputStream*. Para ler o conteúdo de um arquivo utilizando esta CLASSE, deve-se criar um novo OBJETO desta CLASSE e utilizá-lo para ler no arquivo. Entretanto, um OBJETO desta CLASSE só trabalha com leitura de bytes, tornando complexa a leitura de uma linha de um arquivo.

Para auxiliar no processo de leitura, existem CLASSES auxiliares que simplificam a leitura de linhas, como a velha conhecida *Scanner* e a *BufferedReader*, por exemplo. Mas para fazer uso de tal classe, ao instanciar um objeto deste tipo, deve-se utilizar outra classe auxiliar, denominada de *InputStreamReader*. Dessa maneira, são criados ao menos três novos objetos para realizar efetivamente a leitura simplificada de um arquivo. Abaixo é ilustrada a dependência entre estas três classes e um exemplo de uso.




```
File arquivo = new File("meu-arquivo.txt");
FileInputStream input = new FileInputStream( arquivo );
InputStreamReader reader = new InputStreamReader( input );
BufferedReader in = new BufferedReader( reader );
while( in.ready() ){
    String aux = in.readLine();
    System.out.println("line: " + aux);
}
in.close();
```

Caminho de arquivo completo, contudo, específico para Windows (C:)

```
---------------------------------
FileInputStream input = new FileInputStream("C:\\arquivo.txt");
InputStreamReader inputReader = new InputStreamReader( input );
Scanner scan = new Scanner( inputReader );
while( scan.hasNextLine() ){
    String aux = scan.nextLine();
    System.out.println("line: " + aux);
}
in.close();
```

Utilize os métodos hasNextX() para verificar o final do arquivo, sendo X uma das possíveis variantes implementadas, ex: hasNext(), hasNextLine(), hasNextFloat()

Observe no código acima que no primeiro exemplo utilizamos um loop para iterar sobre todas as linhas do arquivo. Na condição deste loop utilizamos o método *ready()* que verifica se o arquivo está pronto para uso e se há algo para ser lido ainda. Outro método que pode ser útil em alguns cenários é o `skip( long n )` que pula *n* caracteres. Já no segundo exemplo, usando o velho e bom Scanner, podemos fazer a leitura como se estivéssemos lendo uma entrada pelo console.

Note que usando um OBJETO da CLASSE Scanner, pode-se ler também valores numéricos usando *nextInt()*, *nextFloat()*, entre outros. Entretanto, lembre-se de duas coisas: (1) verificar se o próximo valor realmente é numérico com o *hasNextFloat()*, por exemplo, a fim de evitar erros, e (2) considerar a possível diferença de representação dos números que depende da linguagem do sistema, usando o auxílio da classe Locale (mencionado na aula-2), caso necessário.

4.4. Outros métodos

Caso seja necessário um melhor desempenho na leitura e escrita em arquivos, duas classes auxiliares podem ser utilizadas: BufferedInputStream e BufferedOutputStream. Estas duas classes utilizam um buffer (em memória) para leitura e escrita, uma vez que o acesso ao disco é muito mais lento do que o acesso a memória.

Outra classe que pode ser utilizada na leitura e escrita de arquivos é a RandomAccessFile. Diferentemente das demais classes apresentadas, esta pode realizar tanto operações de leitura quanto escrita em arquivos. Possui a simplificação do método de leitura *readLine()*, assim como outros métodos de leitura específicos [*readFloat()*, *readDouble()*, *readInt()*].

Exercícios

1) A série de fibonacci tem como seus dois primeiros termos os valores 0 e 1. Os demais termos são construídos pela regra: $f_n = f_{n-1} + f_{n-2}$ sendo *n* o enésimo termo da série. Faça um programa em Java que leia o valor de *n*, calcule e armazene em um vetor os *n+1* primeiros termos da série de Fibonacci. Na sequência, leia dois valores *a* e *b*, sendo $a \leq b \leq n$ e calcule e mostra a soma do a-ésimo (f_a) termo da sequência até o b-ésimo (f_b) termo da sequência. Escreva estas funcionalidades em métodos separados.

2) Receba dois "números inteiros" escritos em uma notação "codificada" e mostre na tela a soma destes valores na mesma notação. Nesta codificação todos os algarismos 1 são trocados pela letra I, todos os algarismos 3 são trocados pela letra E, todos os algarismos 5 são trocados pela letra S e todos os algarismos 0 (zero) são trocados pela letra O.

Ex. Entrada: EI2O 2S97 Ex. Saída: S7I7

3) Implemente uma classe em Java para simplificar e facilitar a criação e extração de informação sobre triângulos. Projete uma classe para triângulos que permite a instanciação de triângulos através de duas formas: passagem de três valores reais que indicam os cantos do triângulo, ou passagem de dois valores reais indicando o cateto oposto e o cateto adjacente de um triângulo retângulo. Elabore métodos de tal forma que seja possível saber se os supostos lados do triângulo formam realmente um triângulo ou não, quais os ângulos internos do triângulo e qual a área interna do triângulo.

Na função principal no programa, gere 10 triângulos aleatoriamente (gerar valores usando a classe Random) usando cada um dos métodos construtores e exiba as informações dos triângulos gerados usando os outros métodos implementados.

4) O arquivo aula-03.txt é um arquivo de resultados de partidas de um esporte desconhecido por humanos. Neste esporte dois times competem um contra o outro e quem marcar mais pontos vence a partida. O arquivo mencionado usa uma forma de preenchimento no estilo csv (*comma separated values*), no qual cada registro aparece em uma linha distinta e os valores são separados por ponto e vírgula. A linha: `Klingon;62;32;Panpum`, por exemplo, indica uma partida entre Klingon vs. Panpum, onde o primeiro time ganhou a partida por 62 a 32. O time vencedor de uma partida recebe dois pontos, o time perdedor recebe uma redução de um ponto e, em caso de empate, ninguém ganha ponto.

Escreva um programa que modele como uma CLASSE estes registros de partidas e implemente duas funções principais: (1) Adicionar novo registro de partida e (2) Mostrar situação do Klingon. Na opção (1) devem ser requisitados e digitados os nomes dos times e o resultado da partida. Para a opção (2) deve-se apenas contabilizar cada jogo do time Klingon a sua devida pontuação, pontos marcados e pontos sofridos. Note que um registro de partida pode ou não ter o time Klingon. Deve-se ignorar os registros referentes a jogos de outros times.