Preparação para Maratona de Informática
"JAVA"

Neste material, veremos como abrir arquivos de texto utilizando alguns recursos Java para manipulação de arquivos, conceitos básicos de manipulação de arrays do Java e formatação de saída.

Como referência, todos arquivos .TXT utilizados devem estar na mesma pasta onde estiver o *script* .java.

Byte Streams

Programas em Java utilizam *byte streams* para implementar entradas e saídas de 8-bits (bytes). Todas as classes byte stream são descendentes das classes InputStream e OutputStream.

Para demonstrar com a byte streams funciona, vamos focar na parte de arquivos de I/O byte streams, FileInputStream e FileOutputStream.

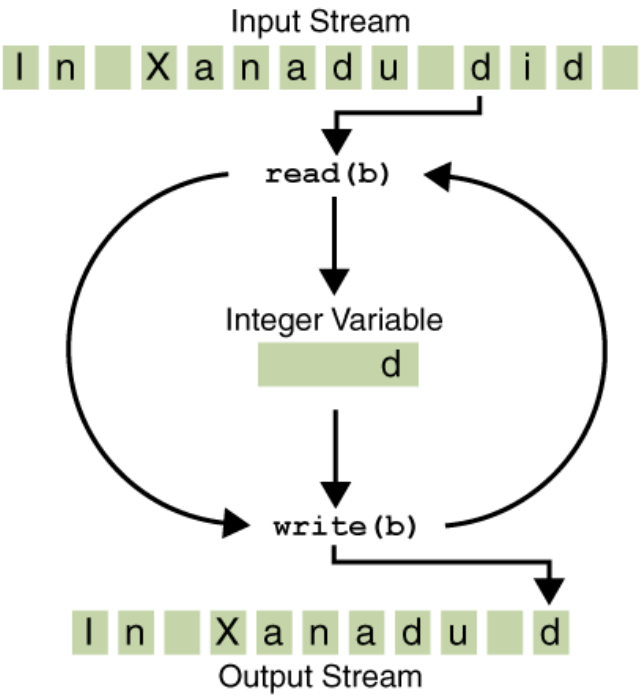
Utilizando Byte Streams

Vamos explorar FileInputStream e FileOutputStream examinando um exemplo de programa chamado CopyBytes, que utiliza byte streams para copiar o arquivo xanadu.txt, byte a byte.

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

public class CopyBytes {
    public static void main(String[] args) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");
```

```
            int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Note que o programa CopyBytes gastou a maior parte do tempo num simples loop que lê uma cadeia de caracteres de entrada e a escreve na saída, um byte de cada vez, como mostrado na figura a seguir:



Simple byte stream input and output.

Perceba que read() retorna um valor int. Assim, utilizando int como valor de retorno, permite que o read() use -1 para indicar que foi alcançado o final do arquivo.

Sempre Fechar Streams

Utilizar sempre o close() no final dos programas de Byte Streams para que tanto a Stream de entrada quanto a de saída, possam ser posteriormente acessadas pelo sistema operacional ou outros programas.

Quando não usar Byte Streams

Byte streams deverá ser utilizado apenas para os mais baixos níveis de acesso de I/O.

Character Streams

A plataforma Java armazena caracteres utilizando a convenção Unicode. Character stream I/O automaticamente traduz este formato interno para o da tabela interna local de caracteres. Em alguns países a tabela de caracteres local geralmente utiliza um conjunto de 8-bits de ASCII.

Um programa que utiliza character streams ao invés de byte streams automaticamente adapta o conjunto de caracteres locais e prontamente faz a internacionalização – tudo isso sem esforço extra do programador.

Usando Character Streams

Todas as classes Stream são decendentes das classes Reader and Writer. Assim como a byte streams, a classe character stream especializa-se em I/O de arquivos: FileReader e FileWriter. O exemplo CopyCharacters ilustra essas classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;

        try {
            inputStream = new FileReader("xanadu.txt");
            outputStream = new FileWriter("characteroutput.txt");

            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
        } finally {
```

```
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters é muito similar ao CopyBytes. A mais importante diferença é que o CopyCharacters utiliza FileReader e FileWriter para entrada e saída no lugar de FileInputStream e FileOutputStream. Perceba que ambos CopyBytes e CopyCharacters utilizam uma variável int para ler e escrever de um arquivo.

Line-Oriented I/O

Caracteres I/O geralmente ocorrem em grandes conjuntos de dados ao invés de um único caractere. Uma unidade comum são as linhas: um conjunto de caracteres (String) com um caractere de final de linha ao final. Um caractere de final de linha pode ser um carriage-return/line-feed sequence ("\r\n") (retorno de carro/pula linha), um simples carriage-return ("\r") (retorno de carro), ou um simples line-feed ("\n") (pular linha ou nova linha). Suportar todas as possibilidades de terminação de linha permite aos programas ler arquivos de textos criados na maior parte dos aplicativos e sistemas operacionais.

Vamos modificar o exemplo CopyCharacters para que utilize line-oriented I/O. Para fazer isso, temos duas classes que não vimos anteriormente: BufferedReader e PrintWriter.

O exemplo CopyLines invoca BufferedReader.readLine e PrintWriter.println para fazer a entrada e saída de uma linha de texto a cada vez.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream =
```

```
                new BufferedReader(new FileReader("xanadu.txt"));
            outputStream =
                new PrintWriter(new
FileWriter("characteroutput.txt"));

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

A utilização do readLine retorna uma linha de texto do arquivo aberto. O CopyLines escreve cada linha utilizando println, que acrescenta um terminador de linha para o sistema atual. Pode acabar não sendo o mesmo terminador de linha que foi utilizado no arquivo de entrada.

Há diversas maneiras de estruturar textos de entrada e saída através de caracteres e linhas. Veremos mais sobre o assunto nas próximas etapas do treinamento.

Durante a Maratona, os alunos receberão exemplos de arquivos de textos de entradas, que deverão ser processadas para a solução dos problemas apresentados, gerando arquivos de textos de saídas, com as soluções encontradas.

Em cada problema apresentado, os nomes dos arquivos de entradas e de saídas, são fornecidos.

Manipulação de Arrays

Vetores (Unidimensionais)

Vetor é uma estrutura que necessita apenas de um índice para a identificação de um elemento nele contido. Sendo assim, para manipular um valor em um vetor é necessário fornecer o nome (identificador) do vetor e o índice do elemento desejado. O índice determina a posição na estrutura onde o elemento está inserido. Cada posição de um vetor contém exatamente um valor que pode ser manipulado individualmente.

Declaração: A declaração de um vetor deverá ser escrita da seguinte forma:

```
String difficult[ ];
double média[ ];
int temp[ ];
```

Outra alternativa de declaração:

```
String[] difficult;
double[] média;
int[]  temp;
```

Criando Objetos Arrays:

Um dos caminhos é usar o operador *new* para criar uma nova instância de um array, por exemplo:

```
int temps[] = new int[9];
```

Quando um objeto array é criado usando o operador *new,* todos os índices são inicializados (0 para arrays numéricos, falso para boolean, '\0' para caracteres, e NULL para objetos). É possível criar e inicializar um array ao mesmo tempo.

```
String nomes[] = new String { "Paulo", "Fernando", "Leticia" , "Carlos", "André"};
```

Cada um dos elementos internos deve ser do mesmo tipo e deve ser também do mesmo tipo que a variável que armazena o array. O exemplo acima cria um array de Strings chamado **nomes** que contém 5 elementos.

Acessando os Elementos do Array:

Uma vez que você tem um array com valores iniciais, você pode testar e mudar os valores em cada índice de cada array.

Os arrays em Java sempre iniciam na posição 0. Por exemplo:

```
String[]  arr= new String[10];
arr[10]="fora do limite";
```

Isto provoca um erro de compilação pois o índice 10 não existe. Como inicia na posição 0, 10 está fora do limite do array.

```
arr[9] = "dentro do limite";
```

Esta operação de atribuição é válida e insere na posição 9 do array, a String “dentro do limite”.

Obtendo o tamanho do array:

```
char [ ] alfabeto = new char [26];
int tamAlfabeto = alfabeto.length;        // tamAlfabeto == 26

String [ ] mosqueteiros = { “ioumle”,“iodoisle”,“iotrêsle” };
int num = mosqueteiros.length;            // num == 3
```

A forma de acesso a um determinado elemento do array é feita simplesmente fazendo referência ao índice do mesmo.

O acesso a um elemento do array é feito colocando uma expressão de valor inteiro entre colchetes após o nome do array.

6.0	7.0	9.0	5.5	9.1	10.0	4.7	7.5	8.6	8.0	nota
0	1	2	3	4	5	6	7	8	9	

nota[3] faz referência ao elemento do vetor, cujo conteúdo é 5.5.

Desta forma, para se ter acesso a qualquer uma das notas armazenadas basta utilizar uma variável inteira qualquer (a título de exemplo a variável **i**) como sendo o índice. Supondo i=5, uma referência a **nota[i], i** seria substituído pelo seu conteúdo no dado instante e neste caso, o elemento referenciado é = 10.0.

O exemplo a seguir cria um array de inteiros chamado tecladoNum e depois preenche o array com inteiros de 0 a 9:

```
int tecladoNum[ ] = new int [10];
for ( int i = 0; i < tecladoNum.length; i++)
    tecladoNum [i] = i;
```

Tentar acessar um elemento fora do intervalo do array gera uma ArrayIndexOutOfBoundsException. Esse e um tipo de RuntimeException, e por isso você pode apanhá-la e tratá-la, ou então ignorá-la:

```
String [ ] estados = new String [10];
try {
       estados[0] = “Pará”;
       estados[1] = “Amazonas”;
       estados[25]=“Amapá”; // Erro: array fora dos limites
      }
```

```
     catch (ArrayIndex0utOfBoundsException erro )
     JOptionPane.showMessageDialog(null, "Erro tratado: " +
erro.getMessage( ));
```

Exemplo 1:

Dado um conjunto com os seguintes elementos: 1, 2, 4, 8, 6, 7, 15, 9, 10, 18. Elabore um aplicativo para calcular a soma desses elementos.

// Calcula a soma dos elementos de um vetor

```
     import javax.swing.*;

     public class SomaVetor {
        public static void main( String args[] )
        {
           int a[] = { 1, 2, 4, 8, 6, 7, 15, 9, 10, 18 };
           int total = 0;
           String saida="Elementos do vetor\n";
           for ( int i = 0; i < a.length; i++ )
              { saida+=a[i]+" ";
                total += a[ i ];}
           JOptionPane.showMessageDialog( null, saida +"\nSoma do
elementos do vetor " + total, "Soma os elementos do vetor",
JOptionPane.INFORMATION_MESSAGE );
           System.exit( 0 );
        }
     }
```

Exemplo 2: Entrada de dados pelo usuário.

//Arquivo ExemploVetor.java
//Cria e permite o preenchimento pelo usuário de um vetor com
//10 elementos do tipo int e ao final mostra seus elementos

```
     import javax.swing.*;
     public class ExemploVetor{
      public static void main (String args[])
        { int v[]=new int[10];

          for (int i=0; i<v.length; i++)
            { v[i]=Integer.parseInt( JOptionPane.showInputDialog(
"Digite o valor do "+(i+1)+
              "° elemento (posição "+i+")")); }
```

```
            String resposta="Posição\tValor";
            for (int i=0; i<v.length; i++)
              { resposta+="\nv["+i+"]\t"+v[i]; }

            JTextArea saida = new JTextArea(11,10);
            saida.setText(resposta);

            JOptionPane.showMessageDialog(null,saida);

             System.exit(0);
      } }
```

Exemplo 3 – Consulta

/*Cria e permite o preenchimento pelo usuário de um vetor com
*10 elementos do tipo int e ao final mostra seus elementos
*e permite a consulta dos valores pela posição */

```
     import javax.swing.*;
     public class ExemploVetorConsulta{
       public static void main (String args[])
         { int v[]=new int[10];

           for (int i=0; i<v.length; i++)
             { v[i]=Integer.parseInt( JOptionPane.showInputDialog(
"Digite o valor do "+(i+1)+ "° elemento (posição "+i+")")); }

           String resposta="Posição\tValor";
           for (int i=0; i<v.length; i++)
             { resposta+="\nv["+i+"]\t"+v[i]; }

           JTextArea saida = new JTextArea(11,10);
           saida.setText(resposta);

           JOptionPane.showMessageDialog(null,saida);

           int consulta = Integer.parseInt(
JOptionPane.showInputDialog( "Digite uma posição válida para
consulta (0-9)"));

           while ((consulta>=0) && (consulta<=9))
            { JOptionPane.showMessageDialog( null,"Posição:
"+consulta+" ---> Valor: "+v[consulta]);
              JOptionPane.showMessageDialog(null,saida);
```

```
                consulta = Integer.parseInt(
JOptionPane.showInputDialog( "Digite uma posição válida para
consulta (0-9)"));
                  }
               System.exit(0);
      }
       }
```

Arrays Multidimensionais

Estruturas indexadas que necessitam de mais que um índice para identificar um de seus elementos são chamadas de matrizes de dimensão n, onde n representa o número de índices requeridos.

Uma matriz de duas dimensões (dois subscritos) pode ser compreendida como uma tabela de valores consistindo em informações organizadas em linhas e colunas. Por convenção, o primeiro subscrito identifica a linha do elemento e o segundo identifica a coluna do elemento. Os arrays que necessitam de dois subscritos para identificar um elemento específico são chamados de arrays bidimensionais.

Exemplo:

	Coluna 0	Coluna 1	Coluna 2	Coluna 3
Linha 0	a [0] [0]	a [0] [1]	a [0] [2]	a [0] [3]
Linha 1	a [1] [0]	a [1] [1]	a [1] [2]	a [1] [3]
Linha 2	a [2] [0]	a [2] [1]	a [2] [2]	a [2] [3]

Declaração: A declaração de uma matriz deverá ser escrita da seguinte forma:

```
int vendas [][];
double notasProvas[][];
```

Arrays multidimensionais podem ser iniciados em declarações da mesma forma que um array unidimendional, isto é:

```
int vendas [][]={{1,2},{3,4}};
```

Essa matriz poderia ser graficamente representada por:

	Coluna 0	Coluna 1
Linha 0	1	2
Linha 1	3	4

Assim como ocorre com arrays unidimensionais, os elementos de um array multidimenional são automaticamente iniciados quando new cria o objeto array.

Exemplo:

```
int b[][];
b=new int [3] []; //aloca linhas
b[0] = new int [5]; //aloca colunas para a linha 0
b[1] = new int [3]; //aloca colunas para a linha 1
```

Exemplo: Dada uma matriz quadrada de ordem M, separar os elementos da diagonal principal em um vetor.

```
import javax.swing.*;

public class DiagonalPrincipal{

public static void main (String args[])
                { int mat[][], diag[],lin,col;


lin=Integer.parseInt(JOptionPane.showInputDialog("Quantas
linhas tem a matriz"));

                    col=lin;
                    mat=new int[lin][col];
                    diag=new int[lin];

                    for (int linha=0; linha<mat.length; linha++)
                         { for (int coluna=0; coluna<
mat[linha].length; coluna++)
                                {
mat[linha][coluna]=(int)(Math.random()*10); }
                    }

                    String resposta="Colunas\t0\t1\t2\t3";
                    resposta+="\nLinhas";

                    for (int linha=0; linha<mat.length; linha++)
                         { resposta+="\n"+linha;
                              for (int coluna=0;
coluna<mat[linha].length; coluna++)
                                { resposta+="\t"+mat[linha][coluna];
}
                    }
```

```
                    resposta+="\n\nDIAGONAL PRINCIPAL";

                    for (int linha=0; linha<mat.length; linha++)
                         {diag[linha]=mat[linha][linha];
                         resposta+="\t"+diag[linha];}

                    JTextArea saida = new JTextArea(resposta);
                    JOptionPane.showMessageDialog(null,saida);
                    System.exit(0);
          }
     }
```

Collection

Como visto no tópico anterior, utilizar arrays é um tanto quanto trabalhoso, a saber:

- um array em Java não pode ser redimensionado em tempo de execução;
- não é possível encontrar diretamente um elemento do qual não saibamos o índice;
- não é possível saber as posições do array que já foram preenchidas, sem criar métodos auxiliares para isso;

- como saber quantas posições já foram usadas no array?

Para resolver essas pendências foi implementado no Java, a partir da versão 2.1.2 a Collections Framework que é um conjunto de diversas classes que implementam estruturas de dados avançadas.

Listas

Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre esses elementos. Isso permite ações como busca, remoção, tamanho “infinito”, entre outras, de maneira mais fácil e ágil.

A implementação mais utilizada da interface List é a ArrayList, que trabalha com um array interno para gerar uma lista.

Neste treinamento abordaremos apenas a ArrayList, que é mais do que suficiente para a resolução dos problemas apresentados na Maratona de Programação.

Vale frisar que existe muita confusão em relação da ArrayList ser um array. Não é. Internamente, ela lança mão de um array como estrutura para armazenar os dados, porém, este atributo está propriamente encapsulado e não há como acessá-lo. Outra dica é que não se pode usar [] com uma ArrayList e nem acessar o atributo length.

Para criar um ArrayList, chama-se o construtor:

```
ArrayList lista = new ArrayList();
```

Pode-se também abstrair a lista da interface List:

```
List lista = new ArrayList();
```

Para criar uma lista de cidades (String), usamos o seguinte código:

```
List cidades = new ArrayList();
lista.add("São Paulo");
lista.add("Ubatuba");
lista.add("Campinas");
lista.add("Caraguatatuba");
```

A interface List possui dois métodos add: um que recebe o objeto a ser inserido e o coloca no final da lista, e um segundo que permite adicionar o elemento em qualquer posição. Perceba que em nenhum momento se faz necessário informar o tamanho da lista. Assim, podemos acrescentar quantos elementos quisermos.

As Listas trabalham de forma genérica. Isto é, não existe ArrayList específica para Strings, outra para Inteiros, etc. Todos os métodos trabalham com Objects.

Podemos então, criar, por exemplo, uma lista de Carros:

```
Carro car1 = new Carro();
car1.marca = "Fiat";

Carro car2 = new Carro();
car2.marca = "Ford";
Carro car3 = new Carro();
car3.marca = "BMW";

Lista carros = new ArrayList();
carros.add(car1);
carros.add(car2);
carros.add(car3);
```

Para saber quantos elementos há na lista, usamos o método size();

```
System.out.println(carros.size());
```

O método get(int) recebe o argumento do índice do elemento que gostaríamos de recuperar da Lista. Assim, podem fazer um for para iterar com a lista de carros:

```
for(int x=0;x<carros.size();x++){
     System.out.println(carros.get(x).getMarca());
}
```

Uma outra maneira, seria recriar um objeto Carro para cada item da Lista:

```
for(int x=0;x<carros.size();x++){
     Carro car = (Carro) carros.get(x);
     System.out.println(car.getMarca());
}
```

Existem outros métodos como o remove(), que recebe um objeto para ser removido da lista; e contains(), que recebe um objeto como argumento e devolve true ou fale, indicando se o elemento está ou não na lista.

A partir do Java 5.0 podemos usar o recurso Generics para restringir as listas a um determinado tipo de objetos:

```
List<Carro> carros = new ArrayList<Carro>();
carros.add(car1);
carros.add(car2);
carros.add(car3);
```

O uso de Generics elimina a necessidade de casting, já que, seguramente, todos os objetos inseridos na lista serão do tipo Carro.

Nesse caso, podemos utilizar um for-each para percorrer a ArrayList Carro:

```
for(Carro car: carros){
     System.out.println(car.getMarca());
}
```

Ao incluir registros em uma lista, os dados ficam disponíveis na ordem em que foram inseridos. Porém, algumas vezes precisaremos percorrer essas listas de forma ordenada.

A classe Collections fornece um método estático sort que recebe uma List como argumento e o ordena por ordem crescete:

```
List<String> cidades = new ArrayList<>();
cidades.add("São Paulo");
cidades.add("Ubatuba");
cidades.add("Caraguatatuba");
```

```
System.out.println(cidades);

Collections.sort(cidades);

System.out.println(cidades);
```

Perceba que na primeira saída a lista é impressa na ordem de entrada, e já na segunda saída ela estará em ordem alfabética.

No entanto, toda lista em Java pode ser de qualquer objeto. Por exemplo, se na classe Carro, tivermos, além da marca, o ano de fabricação, a cor, entre outros dados. E se quisermos ordenar pelo ano de fabricação, como seria ordenada a lista abaixo:

```
Collections.sort(carros);
```

Sempre que formos trabalhar com ordenação de objetos, precisaremos definir um critério de ordenação. Assim, será necessário dizer ao método sort sobre como comparar algum dado do objeto, afim de determinar a ordem da lista.

Assim para ordenar a classe Carro pelo ano de fabricação, basta implementar a interface Comparable, com o método compareTo:

```
public class Carro implements Comparable<Carro>{

     String marca;
     int ano;

     //... o restante do código anterior ficaria aqui

     public int compareTo(Carro outrocarro){
          if(this.ano < outrocarro.ano){
               return -1;
          }
          if(this.ano > outrocarro.ano){
               return 1;
          }
          return 0;
     }
}
```

Nesse caso, toda vez que chamarmos o método sort de Collections, ele saberá como fazer a ordenação da lista. Utilizará o critério que definirmos no método compareTo.

Formatando

Objetos de cadeias de caracteres que contenham a implementação de formatação são instâncias de PrintWriter, uma classe de cadeias de caracteres, e PrintStream, uma classe de controle de bytes.

Dois níveis de formatação estão disponíveis:

- ***print*** e ***println*** formatam valores individuais em um modo padrão;
- ***format*** formata qualquer valor numérico baseado numa String, quando são necessárias várias opções para uma formatação.

Métodos print e println

Os métodos ***print*** ou ***println*** enviam para a saída um simples valor após convertê-lo usando um método ***toString*** apropriado. Podemos ver o que ocorre no exemplo Root:

```
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is "
+ r + ".");
    }
}
```

A saída de Root será:

```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```

As variáveis i e r são formatadas duas vezes: na primeira vez utilizando um código que se sobrepõe ao ***print***, na segunda vez pela conversão automática gerada pelo compilador Java, que também se utiliza do método ***toString***. Você pode formatar qualquer valor dessa forma, porém não terá muito controle dos resultados.

O Método format

O método ***format*** formata múltiplos argumentos baseado numa string de formatação. A string de formatação consiste em um texto estático que segue as especificações de uma formatação. A string formatada não sofre alterações no seu conteúdo, apenas o que será mostrado na saída padrão é que sairá formatado.

A formatação de strings possui várias opções. Cobriremos aqui algumas das mais básicas. Para uma completa descrição, consulte format string syntax nas especificações da API.

O exemplo Root2 formata dois valores com a simples chamada do método ***format***:

```
public class Root2 {
     public static void main(String[] args) {
         int i = 2;
         double r = Math.sqrt(i);

         System.out.format("The square root of %d is %f.%n", i, r);
     }
}
```

Aqui está a saída do Root2:

```
     A raiz quadrada de 2 é 1.414214.
```

Assim como os três formatos usados neste exemplo, todas as especificações de formatação começam com um caractere `%` e terminam com 1 ou 2 caracteres de conversão que especificam o tipo de saída formatada que está sendo gerada. Os três tipos de conversores usados aqui são:

- d formata um valor inteiro para um valor decimal.
- f formata um valor de ponto flutuante (float) como um valor decimal.
- n coloca um terminador de linha de acordo com a plataforma especificada.

Aqui estão algumas outras conversões:

x formata um inteiro para seu valor hexadecimal.

s formata qualquer valor para uma string.

tB formata um inteiro para um nome de mês na plataforma local.

Existem muitas outras conversões.

Nota: Exceto para %% e %n, todas as especificações de formatação devem seguir seus argumentos. Caso contrário acontecerá um erro de exceção.

Na linguagem de programação Java, o comando \n sempre gerará o caractere linefeed (pular linha) (\u000A). Não use \n, ao menos que você queira realmente um caractere de linefeed. Para poder utilizar o caractere correto de pular linha da plataforma local, utilize %n.

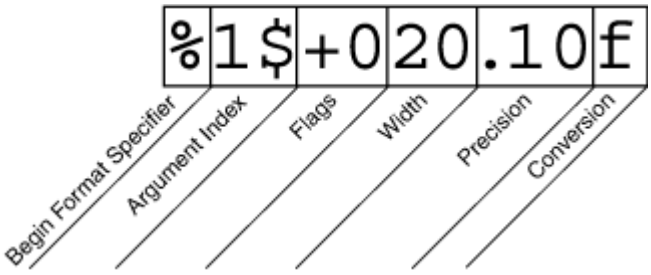
Somando-se as possibilidades de conversão, as especificações de formatos podem conter diversos elementos adicionais que permitirão customizar a saída formatada. Aqui está um exemplo, Format, que utilize todas as possibilidades desse tipo de elementos.

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);
    }
}
```

Eis a saída:

```
3.141593, +00000003.1415926536
```

Os elementos adicionais são opcionais. A seguinte figura mostra como é o posicionamento desses elementos.



Os elementos devem aparecer na mesma ordem mostrada acima. A partir da direita, os elementos opcionais são:

- **Precision**: para valores de ponto flutuante, este é o elemento de precisão do valor formatado. Para “s” e outros tipos comuns de conversão, este é o tamanho máximo de valores formatados; o valor será truncado à direita se necessário.

- **Width**: o tamanho mínimo de um valor formatado; o valor será “acomodado” se necessário. Por padrão o valor é preenchido à esquerda com espaços.

- **Flags**: específica opções adicionais de formatação. No exemplo Format, a flag "+" flag especifica que o número deverá sempre ser formatado com um sinal, e a flag "0" flag especifica que o caractere "0" será o caractere de preenchimento. Outras flags incluem - (ajuste à direita) e, (formata um número com a especificação da plataforma local para separadores de milhares). Note que algumas flags não podem ser usadas com algumas outras flags ou com certos tipos de conversões.

- O **Argument Index** permite explicitar o casamento com um argumento designado. Você também poderá especificar "<" para casar o mesmo argumento com sua prévia especificação. Assim o exemplo ficaria dessa forma:

```
System.out.format("%f, %<+020.10f %n", Math.PI);
```

Método Split

Quebra a string de acordo com a expressão regular dada.

```
public String[] split(String regex,int limit)
```

O array de retorno desse método contém cada substring desta string terminada com outra substring que seja igual à expressão regular fornecida ou que termine com ela. As substrings no array gerado estarão na mesma ordem que aparecem na string original. Se a expressão regular fornecida não for igual à nenhuma parte da string original, o array resultante conterá apenas um elemento com todos os caracteres da string original.

O parâmetro de limite controla o número de vezes que o padrão da expressão regular fornecida será aplicado na string original e afetará a quantidade de elementos do array resultante; Se o limite *n* for maior que zero então o padrão será aplicado em pelo menos *n* - 1 vezes, o tamanho do array não será maior que *n*, e a última entrada conterá todas as entradas que combinarem com o delimitador. Se *n* for negativo, então o padrão será aplicado quantas vezes possível e o array terá um tamanho qualquer. Se *n* for zero o padrão será aplicado quantas vezes possível, o array terá um tamanho qualquer e qualquer espaço vazio nas strings será descartado.

Aplicando o método split na string "boo:and:foo", por exemplo, fornecerá os seguintes resultados com estes parâmetros:

Regex	Limit	Result
:	2	{ "boo", "and:foo" }
:	5	{ "boo", "and", "foo" }
:	-2	{ "boo", "and", "foo" }
o	5	{ "b", "", ":and:f", "", "" }
o	-2	{ "b", "", ":and:f", "", "" }
o	0	{ "b", "", ":and:f" }

Parâmetros:
regex – expressão regular delimitadora

Retorno:
Um array de strings formado pela "quebra" da string original em pedaços (substrings) delimitados pela expressão regular.

Leitura de Arquivo com Split e Gravando o Resultado

Exemplo de Leitura de Arquivo utilizando o método Split (String) para separação de dados das linhas:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

public class Leitura {

    static List<String> vetor = new ArrayList();

    public static void main(String[] args) {

        lerArquivo();
        escreverArquivo();

    }

    static void lerArquivo() {

        Scanner ler = new Scanner(System.in);

        String nome = "Teste.in";

         try {
             FileReader arq = new FileReader(nome);
             BufferedReader lerArq = new BufferedReader(arq);
             System.out.println(nome);
             String linha = lerArq.readLine();
```

```
            while (linha != null) {

                String temp[] = linha.split(";");

                vetor.addAll(Arrays.asList(temp));

                linha = lerArq.readLine();

            }

            arq.close();
        } catch (IOException e) {
            System.err.printf("Erro na abertura do arquivo:
%s.\n",
                    e.getMessage());
        }
    }

    static void escreverArquivo() {

        String nome = "Teste.out";

        try {

            // o parâmetro true/false da linha abaixo serve para
definir
            // se você deseja acrescentar algo no final do
arquivo (true) ou gerar
            // um novo arquivo com o que deseja gravar (false)

            FileWriter arq = new FileWriter(nome, false);

            BufferedWriter escArq = new BufferedWriter(arq);

            for (int i = 0; i < vetor.size(); i++) {
                System.out.println("Escrevendo: " +
vetor.get(i));
                escArq.append(vetor.get(i));
                escArq.newLine();
            }

            escArq.close();

            arq.close();
        } catch (IOException e) {
```

```
                System.err.printf("Erro na abertura do arquivo:
%s.\n",
                        e.getMessage());
            }
        }
    }
```

Percebam que o programa lê um arquivo de texto Teste.in, composto de algumas linhas, com itens separados por pontos-e-vírgulas.

As linhas são “quebradas” nos seus elementos, de acordo com o posicionamento dos pontos-e-vírgulas, e esses elementos são gravados sequencialmente, linha a linha, em um arquivo de saída Teste.out.

Esse será o padrão dos exercícios que serão propostos aos times de alunos na Maratona de Programação.