Funções Matemáticas

A linguagem Java possui uma classe com diversos métodos especializados em realizar cálculos matemáticos. Para realizar esses cálculos, são utilizados os métodos da clase **Math**.

A classe Math define duas constantes matemáticas, sendo **Math.PI** o valor do pi (3.14159265358979323846) e **Math.E** que se refere ao valor da base para logaritmos naturais (2.7182818284590452354).

Os métodos mais comuns da classe Math são:

- **Math.ceil()**
Este método tem como função realizar o arredondamento de um número do tipo double para o seu próximo inteiro.

```
public class ExemploCeil {
 public static void main(String[] args) {
  double a = 5.2,
         b = 5.6,
         c = -5.8;

  System.out.println(Math.ceil(a));  // apresentará 6.0
  System.out.println(Math.ceil(b));  // apresentará 6.0
  System.out.println(Math.ceil(c));  // apresentará -5.0
 }
}
```

- **Math.floor()**
Assim com ceil, o método floor também é utilizado para arredondar um determinado número, mas para seu inteiro anterior.

```
public class ExemploFloor {
 public static void main(String[] args) {
  double a = 5.2,
         b = 5.6,
         c = -5.8;

  System.out.println(Math.floor(a));  // apresentará 5.0
  System.out.println(Math.floor(b));  // apresentará 5.0
  System.out.println(Math.floor(c));  // apresentará -6.0
 }
}
```

- **Math.max()**
Utilizado para verificar o maior valor entre dois números, que podem ser do tipo double, float, int ou long.

```
public class ExemploMax {
 public static void main(String[] args) {
  int a = 10, b = 15;
  double c = -5.9, d = -4.5;

  System.out.println(Math.max(a,b));  // apresentará 15
  System.out.println(Math.max(c,d));  // apresentará -4.5
  System.out.println(Math.max(a,c));  // apresentará 10.0
 }
}
```

- **Math.min()**
O método min fornece o resultado contrário do método max, sendo então utilizado para obter o valor mínimo entre dois números.

```
public class ExemploMin {
 public static void main(String[] args) {
  int a = 10, b = 15;
  double c = -5.9, d = -4.5;

  System.out.println(Math.min(a,b));  // apresentará 10
  System.out.println(Math.min(c,d));  // apresentará -5.9
  System.out.println(Math.min(a,c));  // apresentará -5.9
 }
}
```

- **Math.sqrt()**
Quando há necessidade de calcular a raiz quadrada de um determinado número, utiliza-se o método sqrt, porém o número do qual se deseja extrair a raiz quadrada deve ser do tipo double e o resultado obtido também será um número do tipo double.

```
public class ExemploSqrt {
 public static void main(String[] args) {
  double a = 900, b = 30.25;

  System.out.println(Math.sqrt(a));  // apresentará 30.0
  System.out.println(Math.sqrt(b));  // apresentará 5.5
 }
}
```

- **Math.pow()**
Este método eleva um determinado número ao quadrado ou a qualquer outro valor de potência. Os argumentos e o resultado são do tipo double.

```
public class ExemploPow {
 public static void main(String[] args) {
  double a = 5.5, b = 2;

  System.out.println(Math.pow(a, b));      // apresentará 30.25
  System.out.println(Math.pow(25, 0.5));   // apresentará 5.0
  System.out.println(Math.pow(1234, 0));  // apresentará 1.0
 }
}
```

- **Math.random()**
O método random retorna um número do tipo double com valor entre 0 e 1. Este resultado é gerado randomicamente com uma distribuição aproximadamente uniforme entre 0 e 1.

Para obtermos números aleatórios uma faixa de valores entre 0 e 99, por exemplo, necessitamos multiplicar o resultado de Math.random() por 100.

```
public class ExemploRandom {
 public static void main(String[] args) {

  for(int i = 0; i < 3; i++) {
   int num = (int) ( Math.random() * 100 );
   System.out.println(num);
  }
 }
}
```

Manipulação de Strings

Uma string é um tipo texto que corresponde à união de um conjunto de caracteres. Em Java toda a sequência de caracteres é representada como uma instância da classe String, é constante e o texto contido em um objeto desta classe não pode ser modificado depois de sua criação.

Seus construtores são:

```
char[] arr =  { 'a', 'b', 'c', 'd'};

String a = new String();
String b = new String(arr);
```

```
String c = new String("Java");
```

Por ser um objeto imutável, as Strings podem ser compartilhadas, assim, a instanciação de um objeto desta classe pode ser feita sem a invocação de seu construtor.

```
String d = "Texto";
```

A API java também oferece suporte especial para a conversão de quaisquer tipos de objetos para um objeto da classe **String** , sendo possível obter a representação textual de objetos de quaisquer classes através de seu método **toString()** , que é definido na classe **Object**.

```
Double db = new Double(5.5);
String st = db.toString();
```

A classe **String** também implementa diversas versões de um método **static**, chamado **valueOf()**, através do qual é possível obter uma representação textual de um dado de qualquer um dos tipos primitivos.

```
int it = 5;
String st = String.valueOf(it);
```

Além do exposto acima String tem outros métodos:

- **length()**
O método length é utilizado para retornar o tamanho de uma determinada string, incluindo também os espaços em branco contidos nela.

```
String st = "Texto explicativo";
System.out.println(st.length()); // apresentará 17
```

- **charAt()**
Utilizado para se obter um caractere de determinada string de acordo com o índice informado. Podemos imaginar a String como um array de caracteres cuja primeira posição é 0.

```
String st = "Texto explicativo";
System.out.println(st.charAt(8)); // apresentará p
```

- **toUpperCase()** e **toLowerCase()**
São utilizados para transformar todas os caracteres de uma determinada string em maiúsculas ou minúsculas respectivamente.

```
String st = "Texto Explicativo";
System.out.println(st.toUpperCase()); // apresentará TEXTO EXPLICATIVO
System.out.println(st.toLowerCase()); // apresentará texto explicativo
```

- **substring()**
Retorna uma cópia de caracteres de uma string a partir de dois índices interos. O primeiro argumento indica a partir de que posição se inicia a cópia, o segundo é opcional e especifica a posição em que termina a cópia dos caracteres.

```
String st = "Texto Explicativo";
System.out.println(st.substring(2));    // apresentará   xto Explicativo
System.out.println(st.substring(0,7)); // apresentará   Texto E
System.out.println(st.substring(3,7)); // apresentará   to E
```

Operações com Datas

Operações com datas são frequentes em um grande número de sistemas. São raros, por exemplo, os aplicativos de banco de dados que não necessitam manipular uma única data sequer. Geralmente, esse tipo de aplicativo precisa lidar com datas em diversas de suas funções.

Para compreender como tratar de datas em um programa qualquer, é preciso identificar as operações fundamentais que precisam ser realizadas com elas: a formatação e a conversão. A formatação diz respeito à transformação de uma data em um texto, definindo-se a sua forma de exibição. Para estas operações normalmente são utilizadas um conjunto de classes: Date, Calendar, DateFormat, SimpleDateFormat e Formatter.

Classe Date

A classe Date representa um instante específico no tempo, com precisão de milissegundos, e pode ser utilizada para representar datas. Quando fora criada, essa classe continha duas funções adicionais: permitia a interpretação de datas em termos de ano, mês e dia e também realizava operações de formatação e conversão. Mas os seus métodos não eram adequados à internacionalização e, desde o JDK 1.1 eles foram depreciados. A partir desta data, essa classe passou a ser usada como repositório da data. Agora, deve-se utilizar as classes DateFormat e SimpleDateFormat para realizar operações de formatação e conversão de datas e as classes Calendar e GregorianCalendar para interpretar uma data em termos de suas partes componentes.

Podemos obter a instância de um objeto do tipo Date da seguinte forma:

```
Date agora = new Date();
```

Este construtor cria um objeto Date com a representação da data e hora no momento da construção deste. Internamente um objeto Date armazena a data como um número do tipo long, cujo valor representa a quantidade de milissegundos entre a data atual e a sua data de referência (1/1/1970 00:00h).

Também podemos construir um objeto do tipo Date a partir de uma representação de data no formato long, conforme a descrição acima.

```
Date hoje = new Date(1082345234000L); //  19/04/2004 00:27:14H
```

Embora a construção acima seja possível, provavelmente é melhor utilizarmos outra alternativa para a construção de datas:

```
Calendar c = new GregorianCalendar(2004,Calendar.APRIL,19,00,27,14);
Date hoje = c.getTime();
```

Esta construção tem o mesmo efeito da construção anterior, e é mais fácil seu entendimento.

Os métodos da classe Date são:

- **after()**
Este método recebe um objeto do tipo Date como argumento e retorna um valor booleano indicando se a data representada pelo objeto atual é posterior a data do objeto passado pelo argumento.

```
public class ExemploAfter {
 public static void main(String[] args) {

  Date agora = new Date();

  Calendar c = new GregorianCalendar(2004,
    Calendar.APRIL, 19, 00, 27, 14);
  Date data = c.getTime();

  if(agora.after(data)) {
   System.out.println("A data: " + agora +
     " é posterior a data: " + data);
  }
 }
}
```

- **before()**
O método before recebe um objeto do tipo Date como argumento e retorna um valor booleano indicando se a data representada pelo objeto atual é anteriot a data do objeto passado pelo argumento.

```
public class ExemploBefore {
 public static void main(String[] args) {

  Date agora = new Date();

  Calendar c = new GregorianCalendar(2004,
    Calendar.APRIL, 19, 00, 27, 14);
  Date data = c.getTime();

  if(agora.before(data)) {
   System.out.println("A data: " + agora +
     " é anterior a data: " + data);
  }
 }
}
```

- **getTime()**
Este método retorna um número do tipo long, cujo valor representa a quantidade de milissegundos entre a data atual e a sua data de referência (1/1/1970 00:00h).

Com este valor podemos calcular a diferença entre datas, pois, se subtrairmos o valor em milissegundos de duas datas, e dividirmos por 1000 milissegundos / 60 segundos / 60 minutos / 24 horas, obteremos o número de dias que representa esta diferença.

```
long agoraMilisec = agora.getTime();
long dataMilisec = data.getTime();
long dias = (agoraMilisec - dataMilisec) / 1000 / 60 / 60 / 24;
```

Classe Calendar e GregorianCalendar

As classes Calendar e GregorianCalendar oferecem mecanismos adequados para realização de cálculos com datas ou para identificação das propriedades de uma data, como, por exemplo, para identificar o dia da semana, o dia do mês em relação ao ano etc.

Para isso estas classes convertem um tipo Date armazenando internamente em um série de campos. Elas também possuem métodos para recuperar (get) e armazenar (set) os valores correspondentes a datas e horas, por meio de um argumento fornecido que identifica o campo a ser manipulado. Por exemplo, para recuperar o dia do ano, pode ser usada a sintaxe **get(Calendar.YEAR)**. Neste caso, a sintaxe define que o campo a ser manipulado é o ano (YEAR) da data.

Os principais campos usados são mostrados na tabela a seguir.

Campo	Descrição
DAY_OF_MONTH	Dia do mês (1 a 31)
DAY_OF_WEEK	Dia da semana (0 = domingo, 6 = sábado)
DAY_OF_WEEK_IN_MONTH	Semana do mês (1 a 5) corrente. Diferente em relação a WEEK_OF_MONTH porque considere apenas a semana cheia
DAY_OF_YEAR	Dias decorridos no ano corrente
HOUR	Hora do dia (manhã ou tarde) (0 a 11)
HOUR_OF_DAY	Hora do dia (0 a 23)
MILLISECOND	Milissegundos em relação ao segundo corrente
MINUTE	Minutos em relação à hora corrente
MONTH	Mês em relação ao ano corrente
SECOND	Segundos em relação ao minuto corrente
WEEK_OF_MONTH	Semana em relação ao mês corrente (1 a 5)
WEEK_OF_YEAR	Semana em relação ao ano corrente
JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER	Mês correspondente ao ano

Campo	Descrição
MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY	Dia correspondente à semana

Para obtermos uma instância de Calendar necessitamos chamar o método getInstance() de Calendar ou utilizar um dos construtores de GregorianCalendar.

```
Calendar agora = Calendar.getInstance();
Calendar umaData = new GregorianCalendar(2010,Calendar.MAY, 20);
Calendar outraData = new GregorianCalendar();
```

A seguir são apresentados os métodos mais utilizados das classes Calendar e GregorianCalendar.

- **add()**
Este método recebe como argumento o campo de Calendar a ser alterado e um valor representando a quantidade de tempo a ser adicionado ou removido.

```
public class ExemploCalendar {
 public static void main(String[] args) {

  Calendar data = new GregorianCalendar(2010, Calendar.MAY, 20);

  data.add(Calendar.DAY_OF_MONTH, -12);
  data.add(Calendar.MONTH, 6);

  // Apresentará  Mon Nov 08 00:00:00 BRST 2010
  System.out.println(data.getTime());
 }
}
```

- **after(), before()** e **equals()**
São métodos equivalentes aos da classe Date, exceto que só funcionam com argumentos do tipo Calendar.

```
public class ExemploCalendar {
 public static void main(String[] args) {
  Calendar agora = Calendar.getInstance();
  Calendar data = new GregorianCalendar(2010, Calendar.MAY, 20);

  if(data.after(agora))
   System.out.println("A data: " + data.getTime() +
      " é superior a data de hoje");
  else if(data.before(agora))
   System.out.println("A data: " + data.getTime() +
   " é anterior a data de hoje");
  else if(data.equals(agora))
   System.out.println("A data: " + data.getTime() +
   " é igual a data de hoje");
  else
   System.out.println("Argumento inválido");

 }
}
```

- **getTime()**
Retorna um objeto do tipo Date.

```
Calendar c = new GregorianCalendar(2010, Calendar.MAY, 20);
Date  data = c.getTime();
```

- **getTimeMillis()**
Este método é equivalente ao método Date.getTime() e retorna o número do tipo long de milissegundos referente a diferença entre a data representada por este objeto Calendar e 1/1/1970 00:00:00h.

Classe DateFormat

A classe DateFormat permite a conversão de uma string com informações sobre uma date em um objeto do tipo Date e também permite apresentar a data com diferentes formatações, dependendo das necessidades de utilização, tornando sua visualização mais agradável aos usuários.

Ao criar um objeto a partir da classe DateFormat, ele conterá informação a respeito de um formato particular no qual a data será apresentada. Existem diferentes formatos aceitos para a classe DateFormat apresentados na tabela a seguir.

Nome do Formato	Exemplo
default	10/12/2010
DateFormat.SHORT	10/12/10
DateFormat.MEDIUM	10/12/2010

Nome do Formato	Exemplo
DateFormat.LONG	10 de Dezembro de 2010
DateFormat.FULL	Sexta-feira, 10 de Dezembro de 2010

- **format()**
Método utilizado na conversão de um objeto do tipo Date para String.

```
DateFormat  df = DateFormat.getDateInstance(DateFormat.FULL);
Date hoje = new Date();
String data = df.format(hoje);

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

- **parse()**
Este método é utilizado para converter uma String contento uma data em um objeto do tipo Date. Sempre que utilizar este método será necessário o tratamento da exceção ParseException.

```
public class ExemploCalendar {
 public static void main(String[] args) {
  try {
                                    // Formato default
    DateFormat  df = DateFormat.getDateInstance();
    String data = "15/01/2011";
    Date hoje = df.parse(data);

    System.out.println(hoje);

  } catch(ParseException ex) {
    System.out.println("Data inválida!");
  }

 }
}
```

Classe SimpleDateFormat

Ela permite criar formatos alternativos para a formatação de datas e horas, dependendo das necessidades do desenvolvedor, ou seja, essa classe possibilita expandir as capacidades da classe DataFormat.

Para que o seja criado o formato de data/hora deve ser montada uma String utilizando os caracteres apresentados na tabela a seguir.

Caractere	Descrição	Exemplo
G	Designador da era	AD
y	Ano	2005 ou 05

Caractere	Descrição	Exemplo
M	Mês do ano	Jul ou 07
w	Semana do ano	15
W	Semana do mês	3
D	Dia do ano	234
d	Dia do mês	5
F	Dia da semana no mês	2
E	Dia da semana	Sex
a	AM ou PM	AM
H	Hora do dia (0-23)	0
k	Hora do dia (1-24)	23
K	Hora do dia (0-11)	2
h	Hora do dia (1-12)	5
m	Minuto da hora	10
s	Segundos do minuto	30
S	Milissegundos	978

As letras apresentadas na tabela acima permitem criar os mais diversos tipos de formatação, sejam numéricos ou textuais. Dependendo da combinação de letras e das quantidades de cada letra usada, são criados os mais diversos formatos. Observe que uma simples alteração de minúscula para maiúscula pode gerar um resultado totalmente diferente.

O padrão para a formatação pode ser informado via construtor da classe SimpleDateFormat ou via o método applyPattern().

Para converter String em objeto do tipo Date ou de objeto do tipo Date para String basta utilizar os métodos parse() e format() respectivamente. Eles são equivalentes aos métodos da classe DateFormat.

```
public class ExemploSimpleDateFormat {
 public static void main(String[] args) {
  try {
   SimpleDateFormat  sdf = new SimpleDateFormat();
   sdf.applyPattern("dd/MM/yyyy");

   String data = "15/01/2011";
   Date hoje = sdf.parse(data);

   sdf.applyPattern("EEEE, 'dia' d 'de' MMMM 'de' yyyy");

   System.out.println(sdf.format(hoje));

  } catch(ParseException ex) {
   System.out.println("Data inválida!");
  }

 }
}
```

Classe NumberFormat

A classe NumberFormat é a classe abstrata base para todos as formatações numéricas. Esta classe fornece a interface para a formatação e interpretação de números.

Use getInstance() ou getNumberInstance() para obter o formatador padrão para números. Use getIntegerInstance() para formação de inteiros. O método getCurrencyInstance() para formatação de valores monetários. E getPercentInstance() para formatação de percentagem.

Classe DecimalFormat

DecimalFormat é a subclasse concreta de NumberFormat que formata números decimais. Ela têm uma variedade de recursos para suportar diferentes tipos de números incluindo inteiros, números com ponto flutuante, notação científica, percentagem e valores monetários.

A classe DecimalFormat utiliza um conjunto de padrões e símbolos para especificar sua formatação, conforme tabela abaixo:

Símbolo	Significado
0	Dígitos
#	Dígitos, na ausência é apresentada zero
.	Separador decimal
-	Sinal negativo
,	Separador de centena
E	Separa a mantissa e expoente na notação científica
;	Separa os padrões para valores positivos e negativos
%	Multiplica por 100
\u2030	Multiplica por 1000
\u00a4	É substituído pelo simbolo monetário
'	Utilizado para marcar os textos que não devem ser considerados como padrão

```
public class ExemploDecimalFormat {
 public static void main(String[] args) {
    try {
     DecimalFormat df = new DecimalFormat("#,##0.00;-#,##0.00");

     String s = "1.654,1";
     Number num = df.parse(s);
     double d = num.doubleValue() * -1;

     df.applyPattern("\u00a4 #,##0.00;\u00a4 (#,##0.00)");

     System.out.println(df.format(d)); // Apresentará  R$ (1.654,10)
    } catch(ParseException e) {
     e.printStackTrace();
    }

 }
}
```

Classe Formatter

Esta classe oferece suporte a leiautes justificados e alinhados, formatação para números, Strings, data e hora para uma localidade (Locale) específica. Tipos Java tais como, byte, BigDecimal, e Calendar são suportados.

```
// Exemplo de Formatação numérica
  Formatter fmt = new Formatter();
  double d = -1654.1;
  System.out.println(fmt.format("R$ %(,.2f", d)); // Apresentará  R$ (1.654,10)

// Exemplo de Formatação de Data
  Date d = new Date();
 // Apresentará  24/11/2010
  System.out.println(fmt.format("%1$te/%1$tm/%1$tY\n", d));
 // Apresentará  Quarta-feira, dia 11 de Novembro de 2010
  System.out.println(fmt.format("%1$tA, dia %1$tm de %1$tB de %1$tY", d));
```

A formatação a ser aplicada é especificada por uma String de formatação. A String de formatação contém uma sequencia de mnemônicos iniciados com o caractere“%”.

A sequencia de mnemônicos para formatar caracteres e tipos numéricos têm o seguinte formato:

%[índice do argumento$][indicador][tamanho][.precisão]conversão

A sequencia de mnemônicos para formatar data e hora têm o seguinte formato:

%[índice do argumento$][indicador][tamanho]conversão

O índice do argumento é opcional e indica a posição do argumento a ser formatado.

O indicador é opcional e é um caractere que modifica a formatação.

O tamanho é um número não negativo que indica a quantidade mínima de caracteres a ser utilizada na formatação.

A precisão é opcional e indica através de um número não negativo a quantidade de dígitos apresentada na posição depois da vírgula.

A conversão é um caractere obrigatório que determina o tipo da formatação aplicada.

Abaixo é apresentada uma tabela com os caracteres mais utilizados na conversão de caracteres, tipos numéricos, datas e horas.

Conversão	Descrição
s ou S	Invoca toString() do argumento
c ou C	É aplicado em caracteres
d	É utilizado em números inteiros
e ou E	É utilizado em números de ponto flutuante e apresenta notação científica
f	É utilizado em números de ponto flutuante
t ou T	É utilizado como prefixo em formatação de data e hora
n	Insere uma nova linha na formatação

A tabela a seguir têm relacionado os subtipos para a formatação de data e hora.

Conversão	Descrição
H	Hora do dia na faixa. 00-23
I (i maiúsculo)	Hora do dia no formato 12 horas na faixa 01-12
k	Hora do dia na faixa 0-23, sem acréscimo de zeros adicionais
l (L minúsculo)	Hora do dia no formato 12 horas na faixa 1-12, sem acréscimo de zeros adicionais
M	Minuto na hora ex. 00 – 59
S	Segundos no minuto ex. 00-60
L	Milissegundos ex. 000- 999
p	Marcados am ou pm
B	Nome do mês ex. Janeiro
b	Nome do mês abreviado ex. Jan
A	Nome da semana ex. Domingo
a	Nome da semana abreviado ex. Dom
Y	Ano com quatro dígitos ex. 1999
y	Ano com dois dígitos ex. 99

Conversão	Descrição
m	Número do mês com 2 dígitos ex. 02
d	Dia do mês com 2 dígitos ex. 01 – 31
e	Dia do mês ex 1 – 31

A tabela a seguir apresenta os indicadores utilizado em formatações

Conversão	Descrição
-	Alinha o resultado à esquerda
+	Sempre apresentará sinal
(espaço em branco)	Acrescentará espaço em valores positivos
0 (número zero)	Preenchimento com zeros em números
,	Apresentará ponto nas centenas
(	Circundará com parenteses números negativos

As classes String e OutputStream também implementam a formatação, tornando possível a utilização do método format diretamente.

```
// Apresentará  24/11/2010
System.out.format("%1$te/%1$tm/%1$tY%n", new Date());
```

Classe Pattern

Uma representação compilada de uma expressão regular.

Uma expressão regular descrita como uma String, deve ser compilada numa instancia desta clase. O padrão resultante pode ser utilizado para criar um objeto do tipo Matcher que identifica o padrão compilado em uma sequencia de caracteres arbitrario.

O método matches() oferece a conveniência de processar uma expressão regular diretamente. Este método compila a expressão recular e efetua a identificaão do padrão resultante contra a String fornecida.

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

A tabela abaixo apresenta os construtores para os padrões regulares mais usuais.

Construtor	Identifica
[abc]	a, b ou c
[^abc]	Qualquer caractere exceto a, b e c
[a-z]	Todos os caracteres de a até z
[a-zA-Z]	Todos os caracteres de a até z e de A até Z
.	Qualquer caractere

\d	Dígitos de 0 até 9
\w	Equivalente [a-zA-Z_0-9]
X?	Ocorrência de X, uma ou nenhuma vez
X*	Ocorrência de X, zero ou mais vezes
X+	Ocorrência de X, uma ou mais vezes
X{n}	Ocorrência de X, exatamente n vezes
X{n,}	Ocorrência de X, pelo menos n vezes
X{n,m}	Ocorrência de X, pelo menos n vezes mas não mais que m
XY	X seguido de Y
X|Y	Ou X ou Y

Abaixo temos um exemplo da utilização de expressões regulares para validar o código do produto no formato “9999-99”:

```
String st = JOptionPane.showInputDialog(
          "Informe o Cod. Produto");

if(st != null && st.matches("[0-9]{4}-[0-9]{1,2}"))
  System.out.println("Cod. Produto válido!");
else
  System.out.println("Cod. Produto inválido!");
```

Tipos Enumerados (enum)

A partir da versão 5.0 do JDK é suportado tipos enumerados conhecidos como enum. Na sua forma mais simples as enumerações se parecem como uma lista de valores:

```
public enum Estacoes {
     PRIMAVERA,
     VERAO,
     OUTONO,
     INVERNO;
}
```

Sua aparência pode enganar. Em Java enumerações são muito mais poderosas. Elas definem:

- A enum define um tipo da mesma forma que class define;
- Limita os valores possíveis para este tipo criando um namespace que agrupa todos os valores possíveis para este tipo;
- As enumerações são definidas como constantes impedindo alterações e possibilitando a uniformidade na sua utilização;
- A impressão de uma enumeração resulta em uma String possibilitando que possamos converter enumerações para String com toString() e String para enumerações com valueOf();
- Uma enumeração implementa Comparable e Serializable.

Podemos adicionar comportamento a enumerações, como no exemplo abaixo

```
public enum Operacao {
  SOMA {
     public double execute(double x, double y) {
       return x + y;
     }
  },
  SUBTRAI {
     public double execute(double x, double y) {
       return x - y;
     }
  },
  MULTIPLICA {
     public double execute(double x, double y) {
       return x * y;
     }
  },
  DIVIDE {
     public double execute(double x, double y) {
       return x / y;
     }
  };

  public abstract double execute(double x, double y);
}
```

Cada enumeração do tipo **Operacao** tem seu próprio comportamento, mas compartilham o mesmo nome. Assim SOMA.execute() efetua a soma de dois double e DIVIDE.execute() divide o valor de dois double. Conforme exemplo abaixo:

```
double x = 4;
double y = 2;
// Apresentará os valores abaixo
// 4,000000 SOMA 2,000000 = 6,000000
// 4,000000 SUBTRACAO 2,000000 = 2,000000
// 4,000000 MULTIPLICACAO 2,000000 = 8,000000
// 4,000000 DIVISAO 2,000000 = 2,000000

for (Operacao op : Operacao.values())
     System.out.printf("%f %s %f = %f%n",
               x, op, y, op.execute(x, y));
```

Toda a enumeração tem definido automaticamente os métodos:

- equals() - compara duas enumerações e retorna true ou false;
- toString() - retorna uma String representando a enumeração;
- valueOf() - converte uma String em enumeração;
- values() - retorna um array das enumerações declaradas (vide exemplo acima).

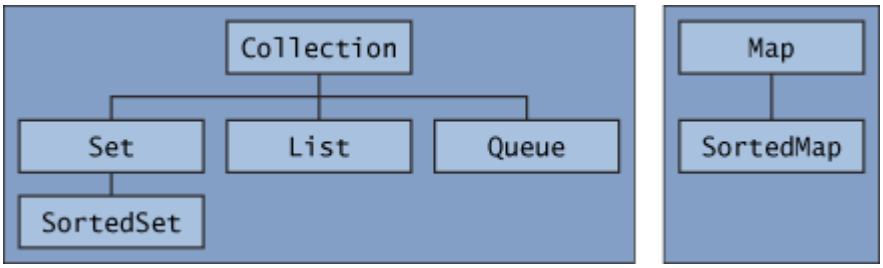
Utiliza enumerações sempre que necessitar de um conjunto fixo de constantes, tais como, sexo, tipos de telefone, estados do pais, por exemplo.

Coleções

O Java Collections Framework é a composição é uma arquitetura unificada para representar e manipular coleções e contém:

- Interfaces: Estas são abstrações de tipos de dados que representam coleções. As interfaces permitem que as coleções sejam manipuladas independentemente dos detalhes de sua implementação;
- Implementações: São as implementações concretas das interfaces. Em essencia são estruturas de dados reutilizáveis;
- Algoritmos: São métodos que executam operações úteis, tais como pesquisa e ordenação, em objetos que implementam as interfaces de coleções.

As coleções encapsulam diferentes tipos de coleções, exibidas na figura abaixo:



Estas interfaces permitem que as coleções sejam manipuladas independentemente dos detalhes de suas implementações. As interfaces são a base do Java Collections Framework.
Um Set é um tipo especial de Collection, um SortedSet é um tipo especial de Set, assim por diante. Note também que a hierarquia consiste em duas arvores distintas. Um Map não é uma Collection.

Ainda todas as interfaces de coleções são genéricas. Veja abaixo a declaração da interface Collection:

```
public interface Collection<E> ...
```

A sintaxe “<E>” informa que a interface é genérica. Quando você declara uma instância de Collection você pode e deve especificar o tipo do objeto que será contido na coleção. Desta forma é possível que o tipo dos objetos adicionados nesta coleção possam ser verificados no momento da compilação.

Abaixo segue a descrição das principais interfaces de Collection.

- Collection – A interface topo da hierarquia de coleções. Uma Collection representa um grupo de objetos chamados de elementos. A interface Collection é o último denominados comum para todas as implementações de coleções e é utilizada como argumento quando é necessário o máximo de generalização.
  Alguns tipo de coleções permitem duplicidade de elementos, e outros não. Alguns são ordenados e outros não. As mais específicas sub-interfaces são Set, List e Queue. Abaixo segue uma lista de alguns métodos declarados nesta interface:

métodos	Descrição
size()	Quantos elementos estão contidos na coleção
isEmpty()	true se a coleção estiver vazia
contains()	Verifica se um objeto está na coleção
add()	Adiciona um elemento à coleção
remove()	Remove um elemento à coleção
clear()	Remove todos os elementos de uma coleção
toArray()	Retorna um array dos elementos contidos na coleção

– Set – Uma coleção que não pode conter duplicidade de elementos. Esta interface modela a abstração matemática de um set, tais como um conjunto de cartas em um jogo de truco, as disciplinas de um curso superior.
As implementações de Set são **HashSet**, que armazena os elementos em uma tabela hash, é sua implementação mais rápida, no entanto não garante a ordem dos elementos. **TreeSet**, que armazena os elementos em uma arvore balanceada, os elementos são ordenados com base em seus valores. É mais lento que o HashSet. **LinkedHashSet**, que implementa uma tabela hash com uma lista ligada, permitindo ordenação de seus elementos baseado na ordem de inserção.

```
// Declara um set de Strings
Set<String> lista = new TreeSet<String>();

// Inclui objetos no set
lista.add("abc");
lista.add("def");
lista.add("fgh");

// Ordena o set
// A implementação TreeSet ordena automaticamente na
// inserção e não permite duplicidade de chave

// Apresenta o set com foreach
for (String txt : lista)
        System.out.println(txt);

// Testa da existencia do objeto no set
if(lista.contains("def"))
        // remove o objeto do set
        lista.remove("def");

// Apresenta o set com while
Iterator<String> it = lista.iterator();
while(it.hasNext()) {
        String txt = it.next();
        System.out.println(txt);
}
```

– List – Uma coleção ordenada (as vezes chamada de sequencia). List pode conter elementos duplicados. Ao adicionarmos um elemento em List é associado um indice a este elemento (sua posição). As implementações mais utilizadas de List são **ArrayList** e **Vector**.

```
// Declara um list de Strings
List<String> lista = new ArrayList<String>();

// Inclui objetos na lista
lista.add("abc");
lista.add("def");
lista.add("fgh");

// Ordena a lista
Collections.sort(lista);

// Apresenta a lista com foreach
for (String txt : lista)
        System.out.println(txt);

// Testa da existencia do objeto na lista
if(lista.contains("def"))
        // remove o objeto da lista
        lista.remove("def");

// Apresenta a lista com for
for (int i = 0; i < lista.size(); i++) {
        String txt = lista.get(i);
        System.out.println(txt);
}
```

– Queue – é uma coleção que armazena elementos em uma FIFO (primeiro que entra é o último que sai). Numa file FIFO, todos os novos elementos são adicionados ao final da fila. Abaixo seguem alguns métodos de queue

métodos	Descrição
remove() e poll()	Remove e retorna o elemento no topo da fila
element() e peek()	Retorna o elemento no topo da fila sem removê-lo
offer()	Usado somente em Queues com limites, adiciona elementos

```
// Declara um queue de Strings
Queue<String> lista = new PriorityQueue<String>();

// Inclui objetos no queue
lista.add("abc");
lista.add("def");
lista.add("fgh");
```

```
// Ordena o map
// A implementação PriorityQueue ordena automaticamente na
// inserção

// Apresenta ao queue com foreach
for (String txt : lista)
        System.out.println(txt);

// Testa da existencia do objeto no map pela chave
if(lista.contains("def"))
        // remove o objeto do map pela chave
        lista.remove("def");

// Apresenta o queue com while
Iterator<String> it = lista.iterator();
while(it.hasNext()) {
        String txt = it.next();
        System.out.println(txt);
}

// Retirando os objetos do queue
for (int i = lista.size(); i > 0; i--) {
        System.out.println(lista.poll());
}
```

– Map – É um objeto que associa valores a chaves. Um Map não pode conter chaves duplicadas: Cada chave só pode estar associada a um valor. Abaixo seguem alguns métodos de map

métodos	Descrição
put()	Adiciona chaves e valores ao Map
get()	Retorna o elemento associado a chave informada
contaisKey()	Retorna true se a chave existe no Map
contaisValue()	Retorna true se o valor existe no Map
values()	Retorna uma Collection dos elementos contidos no Map

```
// Declara um map de Strings
Map<String, String> lista = new TreeMap<String, String>();

// Inclui objetos no map
lista.put("chave1", "abc");
lista.put("chave2", "def");
lista.put("chave3", "fgh");

// Ordena o map
// A implementação TreeMap ordena automaticamente na
// inserção pela chave e não permite duplicidade desta e
// substitui o valor caso isto aconteça
```

```
// Apresenta o map com foreach
for (String txt : lista.values())
        System.out.println(txt);

// Testa da existencia do objeto no map pela chave
if(lista.containsKey("chave2"))
        // remove o objeto do map pela chave
        lista.remove("chave2");

// Apresenta o map com while
Iterator<String> it = lista.values().iterator();
while(it.hasNext()) {
        String txt = it.next();
        System.out.println(txt);
}
```

Operações com Arquivos

Antes de falar a respeito de como efetuar gravações de arquivos em Java é necessária a definição do que é **Streams**.

Streams

Streams são uma abstração de baixo nível para a comunicação de dados em Java. Um Stream representa um ponto num canal de comunicação.

O canal de comunicação normalmente conecta um output stream a um correspondente input stream. Tudo o que é gravado no output stream pode ser lido do input stream. Esta conexão pode ocorrer a partir de um ponto de conexão através de uma rede de computadores, de uma área de memória entre processos, para um arquivo, ou para a console do computador. Os Streams oferecem uma interface padronizado para a transferência de dados para aplicações, independente de qual seja o canal de comunicação utilizado.

Todas as classes que efetuam operações de leitura e escrita (I/O) em Java pertencem ao pakage “java.io”.

FIFO – First in First out (O primeiro a entrar é o primeiro a sair)

Os Streams são uma fila do tipo FIFO. Isto significa que a primeira coisa que for escrita em um output stream será a primeira a ser lida o input stream.

Assim, Streams oferecem um acesso sequencial para os canais de comunicação. Muitas implementações de Streams não oferecem acesso randômico em função do canal de comunicação utilizado.

Uma importante característica dos Streams que que existe o bloqueio do processamento quando é tentada a leitura de um input stream enquanto não existe nada a ser lido. Assim

a única forma da liberação do bloqueio é o envio de uma informação através do output stream corresponde.

Classe File

Esta classe representa o nome de um arquivo independente da arquitetura do Sistema Operacional. Ela oferece vários métodos para determinar informações sobre arquivos e diretórios, bem como permitindo a modificação de seus atributos.

Para obtermos uma instância de objeto File utilizamos um dos seguintes construtores:

```
File f1 = new File("c:\\MeusProjetos");
File f2 = new File("c:\\User\\Fulano\\Desktop", "meuDoc.txt");
File f3 = new File(f1, "Projeto1.doc");
```

Abaixo segue a tabela com alguns dos métodos de File:

Método	Descrição
canRead()	Retorna true se o arquivo pode ser lido
canWrite()	Retorna true se o arquivo pode ser gravado
delete()	Retorna true se obtiver sucesso na deleção do arquivo
exists()	Retorna true se o arquivo existir
getName()	Retorna o nome do arquivo sem o diretório
getPath()	Retorna o nome do arquivo com a parte do diretório
getDirectory()	Retorna o nome do diretório onde o arquivo reside
isFile()	Retorna true se for um arquivo
isDirectory()	Retorna true se for um diretório
length()	Retorna o tamanho do arquivo
list()	Retorna um array de Strings contendo os nomes dos arquivos e sub-diretórios contidos num diretório
mkdir()	Retorna true se conseguir criar o diretório
renameTo()	Retorna true se conseguir renomear o arquivo

A seguir um exemplo de um trecho de código que teste pela existência de um arquivo, criando se necessário, testando se pode gravar e se é um diretório e renomeando um arquivo:

```
if(!f1.exists()) {
        f1.mkdir();
} else if(f1.canWrite() && f1.isDirectory()){
        f3.renameTo(f2);
}
```

Classes FileOutputStream, FileWriter, PrintWriter, FileInputStream, FileReader e BufferedInputStream

As classes FileOutputStream, FileWriter e PrintWriter são utilizadas para a gravação de dados em arquivos. Enquanto FileOutputStream é orientado a byte e oferece os métodos ***write(byte[] b, ...)***, FileWriter e PrintWriter são orientados a caractere e oferecem os métodos ***write(String s, ...)***.

PrintWriter ainda acrescenta métodos tais como ***printf()***, ***print()*** e ***format()*** semalhantes aos utilizados em **System.out**.

Abaixo é apresentado um exemplo de utilização das classes FileOutputStream, FileWriter e PrintWriter, vale notar a necessidade de tratamento de exceção do tipo FileNotFoundException e IOException.

```
try {
        // Declara um objeto do tipo File
        File fl = new File("c:/User/Fulano/Desktop/meuDoc.txt");

        // Abre o arquivo para gravação
        FileOutputStream fo = new FileOutputStream(fl);

        // Grava um texto no arquivo
        String txt = "Texto de exemplo 1\n";
        // Transforma a String em array de bytes
        fo.write(txt.getBytes());

        // Fecha o arquivo
        fo.close();

        // Abre o arquivo para acrecentar mais textos
        FileWriter fw = new FileWriter(fl, true);

        // Grava um texto
        fw.write("Texto de Exemplo 2\n");

        // Cria um PrintWriter a partir do FileWriter
        PrintWriter pw = new PrintWriter(fw);

        // Grava um novo texto
        pw.println("Texto de Exemplo 3");

        // Fecha o arquivo
        fw.close();

} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

As classes FileInputStream, FileReader e BufferedReader são utilizadas para a leitura de dados de arquivos. Enquanto FileInputStream é orientado a byte e oferece o método ***read(byte[] b, ...)***, FileReader e BufferedReader são orientados a caractere. Enquanto FileReader oferece o método ***write(char[] c, ...)***, BufferedReader oferece o método ***readLine()*** que é mais eficiente para ler linhas de arquivos texto.

Abaixo é apresentado um exemplo de utilização das classes FileInputStream, FileReader e BufferedReader, vale notar a necessidade de tratamento de exceção do tipo FileNotFoundException e IOException.

```
try {
        // Declara um objeto do tipo File
        File fl = new File("c:/User/Fulano/Desktop/minhaImagem.dat");

        // Abre o arquivo para leitura
        FileInputStream fi = new FileInputStream(fl);

        // Reserva a área de leitura
        byte[] buffer = new byte[1024];
        // Lê 1024 bytes do arquivo
        fi.read(buffer);
        // Fecha o arquivo
        fi.close();

        // Abre o arquivo para leitura
        FileReader fr = new FileReader("C:/meusDados/lista.txt");

        // Reserva a área de leitura
        char[] texto = new char[1024];
        // Lê uma sequencia de caracteres do arquivo
        fr.read(texto);

        // Cria um PrintWriter a partir do FileWriter
        BufferedReader br = new BufferedReader(fr);

        // Lê todas as linhas restante do arquivo
        String linha = br.readLine();

        while(linha != null) {
                System.out.println(linha);
                linha = br.readLine();
        }

        // Fecha o arquivo
        fr.close();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

Gravação e Leitura de Objetos

A classe ObjectOutputStream é utilizada para gravar objetos em arquivos, este processo é chamado de persistência de objetos. O que ocorre com o objeto é sua serialização, ou seja sua transformação em uma sequencia de bytes para posterior armazenamento.

Toda a informação a respeito da classe que o objeto pertence, sua herança e agregações também é armazenada.

Um objeto para que possa ser serializado deve implementar a interface **Serializable** obrigatoriamente.

Gravamos um objeto utilizando o método ***writeObject()*** de *ObjectOutputStream*, conforme exemplo a seguir.

```
try {
        // Cria um objeto para ser gravado
        List<String> lista = new ArrayList<String>();

        // Adiciona objetos na lista
        lista.add("Texto");
        lista.add("java");
        lista.add("Novo");

        // Cria um arquivo para gravação
        FileOutputStream fo = new FileOutputStream("meusObjetos.dat");

        // Cria um ObjectOutputStream a partir do FileOutputStream
        ObjectOutputStream objOut = new ObjectOutputStream(fo);

        // Grava o objeto
        objOut.writeObject(lista);

        // fecha o arquivo
        objOut.close();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

A classe ObjectInputStream é utilizada para ler objetos armazenados em arquivos. O que ocorre com os bytes lidos é a reconstrução para o objeto de origem.

Toda a informação a respeito da classe que o objeto pertence, sua herança e agregações é utilizada para a reconstrução dos objetos.

Lemos um objeto utilizando o método ***readObject()*** de *ObjectInputStream*, conforme exemplo a seguir.

```
try {

        // Abre um arquivo para leirura
        FileInputStream fi = new FileInputStream("meusObjetos.dat");

        // Cria um ObjectInputStream a partir do FileInputStream
        ObjectInputStream objIn = new ObjectInputStream(fi);

        // Lê o objeto
        List<String> lista = (List<String>)objIn.readObject();

        // fecha o arquivo
        objIn.close();

        // Exibe os objetos lidos
        for(String txt : lista) {
                System.out.println(txt);
        }

} catch(ClassNotFoundException ex) {
        ex.printStackTrace();
} catch(FileNotFoundException ex) {
        ex.printStackTrace();
} catch(IOException ex) {
        ex.printStackTrace();
}
```

Note que quando utilizamos ***readObject()*** necessitamos de um *cast* (List<String> - do exemplo) e que também tratamos a exceção **ClassNotFoundException**.