UNIDADE 6 – Utilização de diversas classes e vetor em Java

Objetivo da Unidade*:

- **Conhecer algumas classes importantes e usuais em Java**
- **Identificar e entender o uso de vetores em Java**

6.1 A classe Object

Todas as classes no Java herdam direta ou indiretamente da classe Object; portanto, seus 11 métodos são herdados por todas as outras classes (DEITEL; DEITEL, 2010, p. 258).

Vejamos alguns métodos:

toString(): esse método indica como transformar um objeto de uma classe em uma String. Ele é utilizado, automaticamente, sempre que é necessário transformar um objeto de uma classe em uma String. Na classe Conta, por exemplo, esse método poderia ser definido da seguinte forma:

```
@Override

public String to String() {

        return (“Conta: “ + this.numero);

}
```

Note o uso da notação **@Override** para o método **toString()**. Isso ocorre porque estamos sobrescrevendo **um método definido em Object**.

getClass: retorna a classe de um objeto. Muito utilizado quando se trabalha na criação de ferramentas geradoras de código ou frameworks. Utilizaremos esse método no exemplo da Figura 4.11 para construir o método toString .

equals(): esse método possibilita comparar os valores de dois objetos. Se considerarmos esses objetos iguais, devemos retornar true; caso sejam diferentes, devemos retornar false.

Quando comparamos dois objetos com o operador **==**, na realidade estamos comparando se eles são o mesmo objeto e não se seus valores são iguais. Isso ocorre porque os objetos em Java são ponteiros para espaços de memória. Assim, dois objetos podem ter os mesmos valores em seus atributos e não serem iguais, pois podem apontar para locais diferentes. Dessa forma, para comparar os valores de dois objetos, devemos utilizar o método *equals*.

Por isso que quando queremos comparar *Strings*, por exemplo, utilizamos o método *equals*. A Figura 44 exibe uma implementação do método equals para a classe *Conta*. Nesse método consideramos que duas contas são iguais se são de uma mesma classe e se têm o mesmo número.

```
@Override
public boolean equals(Object o) {
    if (o == null) {
        return false;
    } else if (o.getClass() != this.getClass()) {
        return false;

    } else if (((Conta) o).getNumero() != this.getNumero()) {
        return false;

    } else {
        return true;
    }
}
```

Figura 44: Exemplo de método equals para a classe Conta

No método *equals* apresentado são feitas as seguintes veriicações:

- *if (o == null):* estamos prevendo que se pode tentar comparar um objeto Conta com um valor nulo (variável não instanciada). Como o objeto Conta que acionou o método *equals* está instanciado, ele não pode ser igual a *null*.

- *if (o.getClass() != this.getClass()):* estamos veriicando se o objeto passa do como parâmetro é da mesma classe que o objeto que está invocando o método, ou seja, se estamos comparando duas instâncias da classe *Conta*. Caso os objetos sejam de classes diferentes, consideramos que eles são diferentes.

- *if (((Conta) o).getNumero() != this.getNumero()):* caso os dois objetos sejam do mesmo tipo (*Conta*), então comparamos os valores do atributo numero dessas contas. Se os números são diferentes, consideramos que são contas diferentes; caso contrário, as consideramos iguais.

É importante conhecermos a hierarquia de uma classe para evitarmos replicar códigos de forma desnecessária.

6.2 A classe String

Já estudamos que Java não conta com um tipo primitivo para trabalhar com cadeia de caracteres. Para isso temos em Java a classe **String**.

Para criar uma instância de *String*, não precisamos utilizar o operador *new*, como acontece com as outras classes. Para instanciar um objeto do tipo *String*, basta declarar uma variável desse tipo e iniciá-la com um valor. É importante saber também que objetos da classe String podem ser concatenados utilizando o operador "+".

Para comparar se os valores de duas Strings são iguais, utilizamos o método "**equals**" e não o operador "==" que é utilizado para tipos primitivos.

A classe *String* conta ainda com diversos métodos muito úteis, dentre os quais podemos destacar:

length: retorna o tamanho (tipo int) de uma String.

charAt: retorna o caracter (char) da String que se localiza no índice passado como parâmetro. Vale ressalta que o primeiro índice de uma String é o índice zero.

toUppperCase: retorna uma String com todas as letras maiúsculas a partir da String que chamou o método.

LowerCase: retorna uma String com todas as letras minúsculas a partir da String que chamou o método.

trim: retorna uma String sem espaços em branco no início e no inal dela, a partir da String que chamou o método.

replace: Retorna uma *String* com substrings trocadas, a partir da *String* que chamou o método. As trocas são feitas de acordo com os parâmetros do método: em que aparecer a *substring1* será substituída pela *substring* 2.

valueOf: retorna uma *String* a partir de um valor de outro tipo, como um número por exemplo.

A Figura 45 apresenta um exemplo de programa que utiliza esses métodos da classe String e, a Figura 46 exibe o resultado da execução de tal programa.

```
package exemplos;

public class ExemploString {

    public static void main(String[] args) {
        String nome = "José";
        String frase = "  Meu nome é ";
        String completa = frase + nome; //Concateno 2 Strings formando uma nova
        System.out.println(completa + "!   ");
        System.out.println("O tamanho do nome é: "+ nome.length());
        System.out.println("O caracter da posicao 2 do nome é "+nome.charAt(2));
        System.out.println("Frase Completa toda em maiusculas:"+completa.toUpperCase());
        System.out.println("Substring de 2 a 8: " + completa.subSequence(2,8));
        System.out.println("Tirando os espaços antes e depois da frase completa: " + completa.trim());
        System.out.println("Substituindo José por João na frase completa: " + completa.replace("José", "João"));
    }
}
```

Figura 45: Exemplos de utilização dos métodos de String

```
run:
  Meu nome é José!
O tamanho do nome é: 4
O caracter da posicao 2 do nome é s
Frase Completa toda em maiusculas:  MEU NOME É JOSÉ
Substring de 2 a 8: Meu no
Tirando os espaços antes e depois da frase completa: Meu nome é José
Substituindo José por João na frase completa:   Meu nome é João
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Figura 46: Saída gerada pelo exemplo da Figura 45

6.3 A classe Math

A linguagem Java possui uma classe com diversos métodos especializados em realizar cálculos matemáticos.

Para realizar esses cálculos, são utilizados os métodos da classe Math que devem apresentar a seguinte sintaxe:

```
Math.<nome do método>(argumentos ou lista de argumentos)
```

Não é necessário importar a classe Math, pois o mesmo faz parte do pacote java.lang

A classe Math define duas constantes matemáticas:

- Math.PI – valor de pi (3,14159265358979323846)
- Math.E – logaritmos naturais (2.7182818284590452354)

Método ceil

Tem como função realizar o arredondamento de um número do tipo double para o seu próximo inteiro. Sua sintaxe é a seguinte:

```
· Math.ceil(<valor do tipo double>)
```

Método floor

É utilizado para arredondar um determinado número, mas para seu inteiro anterior. Sua sintaxe é:

```
· Math.floor(<valor do tipo double>);
```

Método max

Utilizado para verificar o maior valor entre dois números, que podem ser do tipo double, float, int ou long. A sua sintaxe é a seguinte:

```
· Math.max(<valor1>,<valor2>);
```

Método min

Fornece o resultado contrário do método max, sendo então utilizado para obter o valor mínimo entre dois números. Do mesmo modo que o método max, esses números também podem ser do tipo double, float, int ou long. A sua sintaxe é a mesma do método max mudando apenas para Math.min

Método sqrt

Utilizado quando há necessidade de calcular a raiz quadrada de um determinado número. O número que se deseja extrair a raiz deve ser do tipo double. Veja sua sintaxe:

```
· Math.sqrt(<valor do tipo double>);
```

Método pow

Assim como é possível extrair a raiz quadrada de um número, também é possível fazer a operação inversa, ou seja, elevar um determinado número ao quadrado ou a qualquer outro valor de potência. Os números utilizados deverão ser do tipo double. Sua sintaxe é a seguinte:

```
· Math.pow(<valor da base>.<valor da potência>);
```

Método random

É utilizado para gerar valores de forma aleatória. Toda vez que o método random é chamado, será sorteado um valor do tipo double entre 0.0 e 1.0 (o valor 1 nunca é sorteado). Nem sempre essa faixa de valores é suficiente numa aplicação real.

Exemplo:

```
int) (Math.random()*100)
```

Com isso seriam gerados números inteiros entre 0 e 99

Exemplo de arredondamento:

Existem algumas formas de arredondar um número fracionado (float e double) transformando-o em um número inteiro e também como obter o valor absoluto de qualquer número..

- abs (número) - retorna o valor absoluto do mesmo tipo do parâmetro (ex.: inteiro retorna int positivo, decimal retorna float positivo, etc)
- ceil (decimal) - este método retorna o valor decimal do parâmetro sem a parte fracionada. Ex.: 2.1 será 2, 6.0 será 6, 10.8 será 10...
- floor (decimal) - este método retorna o primeiro inteiro após o valor decimal. Ex.: 2.1 será 3, 6.0 será 6, 10.8 será 11...
- rint (decimal) - retorna um valor double mais próximo do valor do parâmetro.
- round (decimal) - retorna o arredondamento aritmético do número decimal passado como parâmetro

```
public class Exemplo {

   public static void main(String[] args) {

      float nr = -5.75f;
```

```
        System.out.println("Absoluto: " + Math.abs(nr) +
                "\nInteiro mais baixo: " + Math.ceil(nr) +
                "\nInteiro mais alto: " + Math.floor(nr) +
                "\nDouble mais próximo: " + Math.rint(nr) +
                "\nArredondamento: " + Math.round(nr));
    }
}
```

6.4 A classe DecimalFormat

Os cálculos matemáticos, em especial os que envolvem multiplicação e divisão, podem gerar resultados com muitas casas decimais.Isso nem sempre é necessário e esteticamente correto, pois apresentar um resultado com muitas casas decimais não é muito agradável e legível à maioria dos usuários. Por exemplo: considere duas variáveis do tipo double x=1 e y=6. Ao realizar a divisão de x por y, aparece na tela o resultado 0.16666666666666666. Esse resultado não é o mais adequado para se apresentar. Seria mais conveniente mostrar o resultado com duas ou três casas decimais.

Para realizar a formatação, é necessário definir um modelo de formatação, conhecido pelo nome de pattern. Considere pattern como o estilo de formatação que será apresentado sobre um valor númerico. Para definir o pattern, são usados caracteres especiais.

Caractere	Significado
0	Imprime o digito normalmente, ou caso ele não exista, coloca 0 em seu lugar. Exemplo: Seja as variáveis int x=4, y=32 e z=154, ao usar o pattern “000”, o resultado impresso na tela seria x➔004, y➔032 e z➔154
#	Imprime o digito normalmente, desprezando os zeros à esquerda do número. Exemplo: Sejam as variáveis double x=0,4 e y=01.34, ao usar o pattern “##.##”, o resultado impresso na tela seria x➔.4, y➔1.34
.	Separador decimal ou separador decimal monetário
-	Sinal de número negativo

Figura 47: Uso de caracteres especiais na formatação de classes decimais

A linguagem Java tem como característica ser utilizada no mundo todo. Em função disso, um mesmo software feito em Java pode ser utilizado por usuários espalhados pelo globo.

Cada país ou região adota certos formatos para representação monetária, apresentação de datas,etc. esses formatos são definidos pelo sistema operacional da máquina e ficam armazenados como configurações locais. O separador de casas decimais, por exemplo, pode ser um ponto ou uma vírgula, dependendo da região.

A classe Locale permite identificar certas propriedades da máquina em que o software está sendo executado.

Marcara de formatação	Formato impresso	Descrição
,##0,00	1,242.50	Separa grupo dos milhares com vírgulas, se número menor que um mostra zeros na frente.
$,##0.00;($,##0.00)	($1,535,50)	Números negativos entre parênteses. mostra $
0.#####	1244.5	Se número entre -1 e 1 mostra zero na frente e não mostra zeros no final.

Figura 48: Máscaras de formatação usando DecimalFormat.

```
import java.text.DecimalFormat;

public class Testa {

public static void main(String[] args) {

DecimalFormat df = new DecimalFormat("#,###.00");

System.out.println(df.format(1234.36));

        double valor = 2000.0;

        double vezes = 3.0;

        double prestacao = valor/vezes;
```

```
        DecimalFormat df1 = new DecimalFormat("0.##");

        String dx = df1.format(prestacao);

        System.out.print(dx);

    }
}
```

6.5 A classe SimpleDateFormat

Os recursos de data e hora devem ser suficientemente flexíveis. O uso de datas e horas torna possível a criação de páginas que exibem informações de maneira dinâmica. Existem 11 classes diferentes para manipulação de datas e horas.

As classes disponíveis para a manipulação de data e hora pertencem a três pacotes diferentes

· java.util – Date, Calendar, GregorianCalendar, TimeZone, SimpleTimeZone

· java.text – DateFormat, SimpleDateFormat, FormatSymbols

· java.sql – Date, Time, Timestamp

A classe Date existe em dois pacotes (util e sql), ambos com características e comportamentos diferentes

A diferença básica entre as classes Date, DateFormat, SimpleDateFormat e Calendar é a seguinte:

· **Date** (pacote util) representa um instante de tempo, sem levar em consideração sua representação ou

localização geográfica, com precisão de milisegundos

· **DateFormat** representa um data com formato String de acordo com um determinado fuso horário e calendário

· **SimpleDateFormat** permite a especificação de diferentes formatos para a data

· **Calendar** representar um instante de tempo de acordo com um sistema particular de calendário e fuso horário

Classe Date

Para utilizar uma classe externa, é necessário que ela esteja na mesma pasta da aplicação ou fazemos sua importação

· *import java.util.Date;*

O compilador compreende que data será um objeto declarado a partir da classe Date

· *Date data = new Date( );*

Essa declaração indica que o objeto data será inicializado com a data e hora atuais do sistema(default)

Para marcar o tempo, Java considera o número de milisegundos decorridos desde 1º de janeiro de 1970.

Cada segundo possui 1.000 milisegundos, cada minuto possui 60 segundos, cada hora possui 60 minutos e

cada dia possui 24 horas, para saber o correspondente em dias, basta multiplicar 1000 x 60 x 60 x 24

getTime() – Esse método retorna um inteiro do tipo long que permite representar milissegundos decorridos

O uso de getTime() permite realizar o cálculo entre datas, bastando calcular a diferença entre os milissegundos

Classe DateFormat

A classe *Date* não fornece um mecanismo de controle sobre a formatação de uma data e não permite converter uma string contendo informações sobre uma data em um objeto Date

A classe *DateFormat* permite apresentar a data com diferentes formatações, dependendo das necessidades de utilização, tornando sua visualização mais agradável aos usuários

A classe *DateFormat* pode criar uma data a partir de uma string fornecida

Ao criar um objeto a partir de uma classe *DateFormat*, ele conterá informação a respeito de um formato particular no qual a data será apresentada

O método getDateInstance tem a seguinte sintaxe:

. *getDateInstance(int estilo)*

Ao invocar o método, deve ser passado um número inteiro que define o estilo de formatação

Métodos mais utilizados da classe DateFormat:

· ***Format(Date d)*** – formata a data em uma string de acordo com o estilo utilizado. Retorna uma String

· ***getInstance()*** – Retorna uma data e hora de acordo com o estilo SHORT. Retorna um DateFormat

· ***getDateInstance()*** – Retorna uma data de acordo com o estilo de formatação local. Retorna um DateFormat

· ***getTimeInstance()*** – Retorna um horário de acordo com o estilo de formatação local. Retorna um DateFormat

· ***parse(String s)*** – Converte a string em tipo Date. Retorna um Date

Classe SimpleDateFormat

Permite criar formatos alternativos para a formatação de datas e horas

Em orientação a objeto dizemos que SimpleDateFormat extends DateFormat

Deve-se recorrer ao uso de um pattern para criar o próprio formato de data/hora

Principais letras para criação de patterns:

— G – designador de era. Formato texto. Exemplo: AD

— Y – ano. Formato Year. Exemplo: 2005;05

— M – mês do ano. Formato Month. Exemplo: Jul;07

— W – semana do ano. Formato Number. Exemplo: 15

— W – semana do mês. Formato Number. Exemplo: 3

— D – dia do ano. Formato Number. Ex.: 234

— D – dia do mês. Formato Number. Ex.: 5

— F – dia da semana no mês. Formato Number. Ex.: 2

— E – dia da semana. Formato Text. Ex.: Sex

— A – am/pm. Formato Texto. Ex.: PM

— H – hora do dia(0-23). Formato Number. Ex.: 0

— K – hora do dia(1-24). Formato Number. Ex.: 23

— K – hora em am/pm(0-11). Formato Number. Ex.: 2

— H – hora em am/pm(1-12). Formato Number. Ex.: 5

— M – minuto da hora. Formato Number. Ex.: 10

— S – segundo do minuto. Formato Number. Ex.: 30

— S – milissegundos. Formato Number. Ex.: 978

Métodos mais utilizados da classe SimpleDateFormat

· applyPattern(String p) – Aplica um pattern à data conforme definido na String p. Retorna void

· toPattern() – Fornece o pattern que está sendo usado no formato de data. Retorna uma String

Exemplo:

Classe Date

```
import java.util.Date;

public class Testa_Date {

public static void main(String[] args) {

Date data = new Date();

System.out.println("Data Agora: "+data);
} }
```

Formatando data Atual

```
import java.util.Calendar;
import java.text.DateFormat;
import java.util.Date;

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

Calendar c = Calendar.getInstance();
c.set(2013, Calendar.FEBRUARY, 28);

Date data = c.getTime();
System.out.println("Data atual sem formatação: "+data);

//Formata a data
DateFormat formataData = DateFormat.getDateInstance();
System.out.println("Data atual com formatação: "+ formataData.format(data));

//Formata Hora
DateFormat hora = DateFormat.getTimeInstance();
 System.out.println("Hora formatada: "+hora.format(data));

//Formata Data e Hora
DateFormat dtHora = DateFormat.getDateTimeInstance();
System.out.println(dtHora.format(data)); }
```

Formatação de Datas

```
import java.text.DateFormat;
import java.util.Date;

public class Formatando_Saida_Datas{

        public static void main(String[] args) {
        Calendar c = Calendar.getInstance();
        Date data = c.getTime();
  DateFormat f = DateFormat.getDateInstance(DateFormat.FULL);
```

```
 //Data Completa
 System.out.println("Data brasileira: "+f.format(data));

 f = DateFormat.getDateInstance(DateFormat.LONG);
 System.out.println("Data sem o dia descrito: "+f.format(data));

 f = DateFormat.getDateInstance(DateFormat.MEDIUM);
System.out.println("Data resumida 1: "+f.format(data));

f = DateFormat.getDateInstance(DateFormat.SHORT);
System.out.println("Data resumida 2: "+f.format(data));}}
```

6.6 Array

Como já aprendemos nas disciplinas anteriores, vetor (array) é uma estrutura de dados utilizada para representar certa quantidade de variáveis de valores homogêneos, ou seja, um conjunto de variáveis, todas do mesmo tipo. Em Java podemos criar vetores de tipos primitivos ou de objetos.

```
Declaração: <tipo_do_dado> <nome_do_vetor> [] = new <tipo_do_dado>[quantidade];

Exemplo: int notas[] = new int[30];
```

No exemplo acima, será criado um vetor de 30 inteiros (inicializados com o valor 0). Também é possível inicializar um vetor com um conjunto de valores ao mesmo tempo em que o declaramos.

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

Uma vez que o vetor já foi devidamente criado, sua utilização é idêntica à utilização de vetores em linguagem C, ou seja, acessamos cada elemento pelo seu índice (os índices de um vetor se iniciam do zero). Por exemplo:

```
i.nt numeros[] = new int[3];
numeros[0] = 57;
numeros[1] = 51;
numeros[3] = 37; // Esta linha gera um erro de execução!
```

A última linha do trecho de código acima causa um erro de compilação, pois o índice 3 não existe em um vetor com apenas três elementos.

Para sabermos o tamanho de um vetor, podemos utilizar o atributo “length”. A Figura 47 exibe um exemplo de utilização desse atributo.

```
package exemplos;
import java.util.Scanner;

public class ExemploVetor {

    public static void main(String[] args) {
        float notas[];//Declaro o vetor mas ainda não aloco!
        /*Na próxima linha vou solicitar que o usuário informe a quantidade de
         * dados, lendo essa quantidade para a variável qtd.
         */
        System.out.println("Informe a quantidade de notas a serem digitadas:");
        Scanner scan = new Scanner(System.in);
        int qtd = scan.nextInt();
        //Agora que sei a quantidade vou alocar o vetor do tamanho desejado
        notas = new float[qtd];
        /*Vou fazer um laço para ler todas as notas. Note que usei o atributo
         * length como condição de parada do laço.
         */
        for (int i=0; i<notas.length;i++){
            System.out.println("Digite a nota do aluno numero " + i);
            notas[i]=scan.nextFloat();
        }
        /* Abaixo temos um laço que roda todo o vetor de notas imprimindo na
         * tela todas as notas atribuídas aos alunos.
         */
        for (int i=0; i<notas.length;i++){
            System.out.println("Nota do aluno numero " + i + " foi " + notas[i]);
        }
    }
}
```

Figura 47: Utilização de vetores e do atributo length

REFERÊNCIAS:

CARVALHO, Victor Albani.; TEIXEIRA, Giovany Frossard **Programação Orientada a Objetos.** Disponível em: <http://redeetec.mec.gov.br/images/stories/pdf/eixo_infor_comun/tec_inf/081112_progr_obj.pdf >. Acesso em: 20 jul. 2013.

BORSOI, Beatriz Terezinha.; BRITO, Robison Cris **Linguagem de Programação Comercial. Parte I.** Disponível em: <http://redeetec.mec.gov.br/images/stories/pdf/eixo_infor_comun/tec_inf/081112_progr_obj.pdf >. Acesso em: 20 jul. 2013.