

Iniciação ao Java:

Com orientação a objetos

Márcio Francisco Campos

Versão 11 de outubro de 2006

Índice

INTRODUÇÃO.	4
PARTE ZERO – INTRODUÇÃO.	5
CAPÍTULO 1 – CONCEITOS BÁSICOS.	5
1.1 A LINGUAGEM E A PLATAFORMA JAVA.	5
1.2 INSTALANDO JAVA.	5
1.3 A SINTAXE BÁSICA DO JAVA.	6
1.3.1 Execução de um “programa” Java.	6
1.3.1 Comentários.	7
1.3.2 Tipo de dados e variáveis.	7
1.3.3 Operadores.	8
1.3.4 Controle de fluxo.	11
1.3.5 Laços.	12
1.3.6 Caracteres(char) e Conjunto de caracteres (String).	13
1.3.6 Vetores	14
1.3.7 Exemplos.	14
1.5 A PLATAFORMA JAVA.	16
PARTE UM – CONCEITOS INICIAIS.	18
CAPÍTULO 2 - CLASSES E OBJETOS	18
2.1. CONCEITOS DE CLASSES E DE OBJETOS.	18
2.2. EM JAVA.	19
2.4. EXERCÍCIOS.	21
CAPÍTULO 3 - VARIÁVEIS E MÉTODOS DE OBJETOS E DE CLASSES.	22
3.1. CONCEITOS DE VARIÁVEIS E MÉTODOS DE UM OBJETO E DE UMA CLASSE.	22
3.2. EM JAVA.	22
3.4. EXERCÍCIOS.	26
CAPÍTULO 4 - MENSAGENS E VISIBILIDADE.	27
4.1. CONCEITOS DE MENSAGENS E VISIBILIDADE.	27
4.2. EM JAVA.	27
4.4 EXERCÍCIOS.	31
CAPÍTULO 5 - SOBRECARGA.	32
5.1. INTRODUÇÃO.	32
5.2. EM JAVA.	32
5.4 EXERCÍCIOS.	34
CAPÍTULO 6 – DELEGAÇÃO E COMPOSIÇÃO.	35
6.1 INTRODUÇÃO.	35
6.2. EM JAVA.	35
6.4.EXERCÍCIO.	38
CAPÍTULO 7 - HERANÇA.	39
7.1. INTRODUÇÃO.	39
7.2 EM JAVA.	39
7.4. EXERCÍCIOS.	42
CAPÍTULO 8 – CLASSES ABSTRATAS.	43

8.1. INTRODUÇÃO.	43
8.2. EM JAVA.	43
8.3 EXERCÍCIOS	45
CAPÍTULO 9 - INTERFACES.	46
9.1. INTRODUÇÃO.	46
9.2 EM JAVA.	46
9.3 EXERCÍCIOS	49
CAPÍTULO 10 – ESTUDO DE CASO.	50
10.1 O CASO.	50
PARTE DOIS – INTRODUÇÃO AOS PRINCÍPIOS DE PROJETO OO: PADRÕES.	57
CAPÍTULO 11 – PRINCÍPIOS E PADRÕES.	57
11.1 O PADRÃO “ESPECIALISTA DA INFORMAÇÃO”.	58
11.2 O PADRÃO DE “CRIAÇÃO”.	58
11.3 O PADRÃO “BAIXO ACOPLAMENTO”.	59
11.4 O PADRÃO “ALTA COESÃO”.	60
11.4 O PADRÃO “CONTROLADOR”.	60
11.5 EXEMPLO UM.	60
REFERÊNCIAS BIBLIOGRÁFICAS.	63

Introdução.

Os livros de Java costumam tratar da linguagem em seus principais aspectos sintáticos e semânticos. Sem dúvida este é um aspecto importante para a divulgação desta linguagem. Entretanto, a linguagem Java está intimamente ligada a conceitos de orientação a objetos. Se estes conceitos não estiverem bem entendidos programar Java será um martírio.

Este livro trata do aprendizado da linguagem Java, mas sob os aspectos da orientação a objetos. Desta forma, este livro não é um tratado sobre a linguagem Java. Assim, não se encontrará nestas páginas um guia completo de referência à linguagem. Os conceitos apresentados de Java são aqueles suficientes para introduzir os conceitos respectivos de orientação a objetos.

De maneira geral, o elo principal do documento é o de ensinar Java sob o foco dos conceitos de orientação a objetos e vice-versa. É através desta ênfase que se mostrará como a linguagem Java está concebida. Somente desta forma se poderá aproveitar e entender os conceitos da linguagem.

Na atual versão o livro é composto de duas partes. A primeira para introduzir ao leitor os conceitos iniciais da linguagem. Na segunda parte apresenta-se os conceitos de orientação a objetos.

Destaca-se que este livro não tem por objetivo ensinar a programar. Assim o leitor deve estar familiarizado com programação devendo saber conceitos básicos de tipos, variáveis, laços etc.

Para a realização dos exemplos é necessário a instalação da máquina virtual Java que pode ser baixada de www.java.sun.com e de um ambiente de programação, como por exemplo o SCITE, encontrado em www.scintilla.org.

Parte Zero – Introdução.

Esta parte tem como objetivo apresentar um primeiro contato com a linguagem Java. A tentativa é de apresentar os conceitos sem necessariamente entrar em conceitos de orientação a objetos. É difícil devido a própria natureza da linguagem.

Esta parte considera que o leitor já tenha alguma noção de programação e procura ser uma ponte inicial para as estruturas comuns entre as linguagens de programação. A Parte Um tratará dos conceitos de orientação a objetos, sendo que a partir destes conceitos o leitor pode aproveitar efetivamente o poder desta linguagem.

Capítulo 1 – Conceitos Básicos.

1.1 A linguagem e a plataforma Java.

Em 2005 faz dez anos da linguagem Java. Nos dias de hoje seria diminuir a tecnologia Java a meramente uma linguagem de programação como qualquer outra. O que Java se tornou foi uma verdadeira plataforma de especificação de sistemas. Originalmente pretendia-se que Java fosse empregada na programação de produtos eletrônicos. As linguagens C e C++ não atendiam as necessidades que se desejavam da nova linguagem, pois necessitavam ser compiladas para cada tipo específico de processador. Além do mais o ciclo de vida dos produtos eletrônicos são maiores que os de software e haveria de se manter a compatibilidade retroativa [Hoff, Shaio e Strubck, 1996].

Assim em 1990 James Gosling iniciou o projeto de linguagem de programação apropriada a este tipo de nicho: aparelhos eletrônicos: a linguagem Java foi criada. O anúncio oficial da linguagem foi feito em maio de 1995 na conferência da Sun World em San Francisco.

As características da linguagem Java são sua abordagem para orientação a objetos, compilada e independente de arquitetura de computadores. Estas características possibilitam a esta linguagem ser robusta e extensível.

Atualmente, Java deixou de ser sinônimo de uma linguagem de programação. Pode-se fazer sistemas que rodem em desktop, em rede, para a internet, para a WEB, com processamento distribuído, para rodar em celulares e palms, entre outros.

1.2 Instalando Java.

Para programar em Java é necessário seguir alguns passos para se configurar o ambiente. O primeiro passo é baixar o SDK, ou o Kit de Desenvolvimento de Software Java, e instalá-lo em sua máquina. Faça a instalação correspondente ao seu ambiente computacional (Windows, Linux etc.). Na instalação você escolherá um diretório onde os programas javas residirão, por exemplo: `c:\Java`.

O próximo passo é a escolha de um ambiente de programação. De forma simplificada pode-se optar pelo próprio *bloco de notas* como o ambiente de programação. Para usá-lo digite o programa Java e salve o programa com extensão `java` e entre aspas, exemplo: “`MeuPrimeiroPrograma.java`”.

Ao salvar o seu programa você o encontrará no diretório especificado por você com a extensão Java. Ao compilá-lo, e se houver erros, você encontrará os erros listados. Se não houver erros, o compilador java terá gerado o arquivo com extensão `class`, por exemplo `MeuPrimeiroPrograma.class`. Os arquivos `.class` são os executáveis do Java. Utilize o programa `javac`, no ambiente DOS, para compilar o seu programa, tal como: `javac MeuPrimeiroPrograma.java`

Para executar o programa com extensão `class` use o programa `java`, tal como: `MeuPrimeiroPrograma.class`.

Se os programas `javac` ou `java` não estiverem sendo encontrados você deverá criar ou atualizar a variável de ambiente `CLASSPATH` que indicará o caminho de seu diretório onde o Java foi instalado.

Se preferir, existem outros ambientes mais apropriados para programação tais como o “Scite” e o “Gel”, ambos em software livre. Outros ambientes mais profissionais são o Netbeans e o Eclipse. Neles estes detalhes já estão, em sua maior parte, integrados ao ambiente.

1.3 A sintaxe básica do Java.

A programação básica do Java segue os mesmos princípios de qualquer outra linguagem, tipos, controle de fluxo, repetição entre outros. Em geral a sintaxe Java é muito semelhante a sintaxe da linguagem “C”. A sintaxe de classes e objetos será vista mais adiante, nos próximos capítulos. Assim temos:

1.3.1 Execução de um “programa” Java.

Esta seção ilustra como executar um programa em Java. Na realidade o que se estará fazendo é a execução de uma classe Java, mas por enquanto fica-se como está. Um exemplo de um código java é o que é mostrado no código 11.

```
public class MeuPrimeiroPrograma //1
    public static void main (String[ ] args) { //2
        System.out.println (“este é o meu primeiro programa”; //3
    } //4
} //5
```

Código 1.1 - tipos de variáveis em Java - baseado no Java tutorial

Repare na linha 1 que é definido uma classe em java que é a classe “`MeuPrimeiroPrograma`”. Este programa deverá ser salvo com a extensão `.java` e quando compilado você encontrará no mesmo diretório o arquivo `MeuPrimeiroPrograma.class`.

1.3.1 Comentários.

// comentários até o fim da linha.

/* indicadores d início e de fim de comentários */

1.3.2 Tipo de dados e variáveis.

Uma variável é um item de dado nomeado, com um identificador simbólico. Para cada variável definida deve-se declarar um nome e um tipo. Uma declaração típica em Java é: *tipo nome_da_variável*. Toda variável, quando definida, possui um escopo de existência. Este escopo determina onde a variável pode ser referenciada. O escopo em Java é definido pelo conjunto de chaves { e }. Assim, no exemplo do código 1.1 as variáveis definidas somente existem dentro do função `main`.

Java possui um conjunto de tipos primitivos que estão disponíveis na linguagem. Seguem abaixo uma lista deles:

boolean	corresponde a um valor verdadeiro ou falso.
byte	corresponde a um valor inteiro com sinal de 8 bits.
short	corresponde a um valor numérico com 16 bits.
char	corresponde a um caractere Unicode de 16 bits.
int	corresponde a um valor numérico inteiro com sinal de 32 bits.
float	corresponde a um valor numérico em ponto flutuante de 32 bits.
double	corresponde a um valor numérico em ponto flutuante de 64 bits.
long	corresponde a um valor numérico inteiro com sinal de 64 bits.

O exemplo ilustra a utilização dos tipos em Java.

```
public class DemoVariaveis {
    public static void main(String args[]) {
        // inteiros
        byte umByte = 1; //o maior valor é 127
        short umShort = 55; //o maior valor é 32767
        int umInteger = 127; //o maior valor é 2147483647
        long umLong = 5000L; //o maior valor é 9223372036854775807
    }
}
```

```

// números reais
float umFloat = 13.336F;//o maior valor é 3.40282e+38
double umDouble = 36.725D ;//o maior valor é 1.79769e+308

// outros tipos primitivos
char umChar = 'S';
boolean umBoolean = true;

// mostrando-os
System.out.println("O valor de umByte é " + umByte);
System.out.println("O valor de umShort é" + umShort);
System.out.println("O valor de umInteger é "+ umInteger);
System.out.println("O valor de umLong é " + umLong);
System.out.println("O valor de umFloat é "+ umFloat);
System.out.println("O valor de umDouble é " + umDouble);
System.out.println("O valor de umChar é " + umChar);
System.out.println("O valor de umBoolean é "+ umBoolean);
}
}

```

Código 1.2 - tipos de variáveis em Java - baseado no Java tutorial

1.3.3 Operadores.

Java possui vários tipos de operadores que podem possuir um(ex.: i++), dois (ex.: =) ou três operandos (?). Operadores, quando relacionados de forma apropriada, compõem uma expressão. Nesta sessão três tipos de operadores serão apresentados: os aritméticos e os relacionais e os lógicos.

Os operadores aritméticos são apresentados na tabela 1.1.

Operador	Exemplo	Descrição
+	5 + 4	A expressão 5 + 4 retorna o valor 9.
-	5 - 4	A expressão 5 - 4 retorna o valor 1.
*	5 * 4	A expressão 5 * 4 retorna o valor 20.
/	20 / 5	A expressão 20 / 5 retorna o valor 4.
%	20 % 5	A expressão 20 % 5 calcula o valor do resto da divisão de 20 / 5 que é zero.

Tabela 1.1 - operadores aritméticos

O código 1.2 ilustra o uso de operadores aritméticos. Verifique que quando em uma expressão aritmética pode possuir operandos de tipos numéricos diferentes tais como long, short e int. Neste caso as conversões são feitas de forma apropriada.

```
public class DemoExpAritmetica {
    public static void main(String[] args) {
        //variáveis
        int i = 20;
        int j = 5;
        double x = 13.465;
        double y = 8.11;

        //operadores ariméticos
        System.out.println("Operadores aritméticos");
        System.out.println("    i + j = " + (i + j));
        System.out.println("    x + y = " + (x + y));
        System.out.println("    i - j = " + (i - j));
        System.out.println("    x - y = " + (x - y));
        System.out.println("    i * j = " + (i * j));
        System.out.println("    x * y = " + (x * y));
        System.out.println("    i / j = " + (i / j));
        System.out.println("    x / y = " + (x / y));
        System.out.println("    i % j = " + (i % j));
        System.out.println("    x % y = " + (x % y));

        //misturando tipos
        System.out.println("Misturando tipos");
        System.out.println("    j + y = " + (j + y));
        System.out.println("    i * x = " + (i * x));
    }
}
```

Código 1.3 - operadores aritméticos - baseado no Java Tutorial

Os operadores relacionais compõem expressões booleanas que retornam valores verdadeiros ou falsos. A tabela 1.2 lista os operadores relacionais.

Operador	Exemplo	Descrição
----------	---------	-----------

>	5 > 4	Cinco é maior que quatro, a expressão retorna o valor verdadeiro.
>=	5 >= 4	Cinco é maior que quatro, a expressão retorna o valor verdadeiro. A expressão 4>= 4 também retornaria verdadeira.
<	5 < 4	Cinco é maior que quatro, logo a expressão retornaria falsa.
<=	5 <= 4	Cinco é maior que quatro, logo a expressão retornaria falsa.
==	5 == 4	Cinco não é igual a quatro, logo a expressão retornaria falso.
!=	5 != 4	Cinco não é igual a quatro, logo a expressão retornaria verdadeira.

Tabela 1.2 - operadores relacionais

O código 1.2 ilustra o uso de operadores relacionais.

```
public class DemoExpRelacionais {
    public static void main(String[] args) {
        //variáveis
        int i = 37;
        int j = 42;
        int k = 42;

        System.out.println("    i > j = " + (i > j));    //false
        System.out.println("    i >= j = " + (i >= j)); //false
        System.out.println("    i < j = " + (i < j));    //true
        System.out.println("    i <= j = " + (i <= j)); //true
        System.out.println("    i == j = " + (i == j)); //false
        System.out.println("    i != j = " + (i != j)); //true
    }
}
```

Código 1.4 - operadores relacionais - baseado no Java Tutorial

Os operadores lógicos quando agrupados compõem uma expressão lógica. Uma expressão lógica retorna um valor verdadeiro ou falso. Os operadores lógicos são o E, o OU e o NÃO. Em Java estes operadores são representados pelos símbolos &&, || e ! respectivamente. A tabela 1.3 exemplifica a aplicação destes operadores.

Operador	Exemplo	Descrição
&&	(5 > 4) && (6 > 5)	A expressão retorna verdadeiro, pois ambas as expressões (5 > 4) e (6 > 5)

		são verdadeiras. De outra forma a expressão retornaria falso.
	(5 > 4) (3 > 6)	A expressão retorna verdadeiro, pois pelo menos uma das expressões, (5 > 4), é verdadeira. De outra forma retornaria falso.
!	! (5 > 4)	A expressão retorna falso pois a expressão (5 > 4) é verdadeira

Tabela 1.1 - operadores lógicos

O código 1.4 mostra aplicação destes operadores.

```
public class DemoExpLogicas {
    public static void main(String[] args) {
        //variáveis
        int i = 37;
        int j = 42;
        int k = 42;
        //mostra os valores das expressões
        System.out.println("    i > j = " + (i > j)); //false
        System.out.println("    k >= j = " + (k >= j)); //true
        System.out.println("    j < i = " + (j < i)); //false
        System.out.println("    j <= i = " + (j <= i)); //false
        System.out.println("    i == j = " + (i == j)); //false
        System.out.println("    i != j = " + (i != j)); //true
    }
}
```

Código 1.5 - operadores lógicos - baseado no Java Tutorial

1.3.4 Controle de fluxo.

Considere o código, apenas ilustrativo, 1.2 a seguir.

```
if (booleano) { /* lista de comandos */ }
else if (booleano) { /* lista de comandos */}
else { /*lista de comandos */}
```

Código 1.6 - exemplo de sintaxe do comando if-then-else.

Onde se lê (booleano) entende-se como uma expressão que retorne um valor booleano, por exemplo, a expressão (5>4) retorna valor verdadeiro. Para vê-lo funcionando é necessário executar o código 1.3.

```
public class IfThen { //1
    public static void main(String args[]){ //2
        if (5>4) { System.out.println ("cinco é maior que quatro");} //3
        else {System.out.println ("cinco é menor que quatro");} //4
    } //5
} //6
```

Código 1.7 - usando o comando if-then-else.

1.3.5 Laços.

O comando for possui a seguinte sintaxe:

```
for (expressão; expressão booleana; expressão) { /* lista de comandos
*/ };
```

Código 1.8 - exemplo de sintaxe do comando for.

Para vê-lo funcionando é necessário executar o código 1.5.

```
public class UsodoFor { //1
    public static void main(String args[]){ //2
        for (int i=0; i<10; i++) //3
            { System.out.println("teste do for");} //4
    } //5
} //6
```

Código 1.9 - usando o comando for.

O comando **while** possui dois tipos, a saber:

```
While (booleano) { /*lista de comandos */ } ou
```

```
Do { /* lista de comandos */ } while (booleano);
```

Código 1.10 - exemplo de sintaxe do comando while.

Para vê-lo funcionando é necessário executar o código abaixo.

```
public class WhileUm { //1
```

```

public static void main(String args[]){           //2
    int i= 0;                                     //3
    while (i<10)                                  //4
        { System.out.println("teste do for");    //5
          i++;      }                             //6
    }                                             //7
}                                               //8

```

Código 1.11 - exemplo de sintaxe do comando WhileDo.

ou

```

public class DoWhile {                           //1
    public static void main(String args[]){      //2
        int i= 0;                                //3
        do {                                     //4
            System.out.println("teste do for");  //5
            i++;                                  //6
        }                                        //7
        while (i<10);                            //8
    }                                           //9
}                                               //10

```

Código 1.12 - exemplo de sintaxe do comando DoWhile.

1.3.6 Caracteres(char) e Conjunto de caracteres (String).

Java possui dois outros tipos de variáveis: a que especifica um caracter e que especifica um conjunto de caracteres. O tipo char já foi apresentado na seção acima. O código 1.9 ilustra o uso destes dois tipos. Verifique que o tipo String se incicia com uma letra maiúscula. Será visto na outra parte deste livro que na realidade este tipo é uma Classe Java, mas por enquanto estes conceitos serão deixados para serem expostos mais adiante. Certifique-se, entretanto, de sempre escrever String com letra maiúscula.

```

public class DemoCharString {                   //1
    public static void main(String args[]){     //2
        char umchar= "a";                       //3
        char outrochar= "b";                   //4
        String umaString;                      //5
        umaString = umchar + outrochar;        //6
        // o operador "+" concatena dois caracteres ou um conjunto
        //de caracteres

```

```

        System.out.println ("umchar " + umchar);           //7
        System.out.println ("outrochar " + outrochar);    //8
        System.out.println ("umaString" + umaString);     //9
    }                                                       //10
}                                                           //11

```

Código 1.13 - exemplo de código com tipo char e String.

1.3.6 Vetores

Vetores são uma das formas de se armazenar um conjunto de valores de um mesmo tipo ou de uma mesma estrutura. Em muitos livros costuma-se utilizar o nome em inglês: Array. Em Java a notação de vetores é feita utilizando-se o símbolo de colchetes “[]”. O exemplo abaixo ilustra a definição de vetores:

```

public class DemoVetores {                                //1
    public static void main(String args[]){              //2
        char[ ] alfabeto;                                //3
        String[ ] nomes;                                 //4
        // inicialização dos vetores.
        alfabeto = new char[26];                         // 5
        alfabeto[0] = "a";                               // 6
        nomes = { "marcio", "Jorge", "Vera"}            // 7
        // mostra valores dos vetores
        System.out.println ("alfabeto " + alfabeto[0]); //8
        System.out.println ("nomes " + nomes[0]);       //9
        System.out.println ("nomes" + nomes[1]);       //10
        System.out.println ("nomes" + nomes[2]);       //11
    }                                                     //12
}                                                         //13

```

Código 1.14 - exemplo de declaração de vetores.

Verifique que na linha três e quatro são declarados os vetores. Nas linhas cinco e sete são mostradas algumas das formas de inicialização de um vetor: inicializando-se a estrutura ou com os próprios valores. A linha seis mostra que a primeira posição de um vetor em Java corresponde ao valor zero.

1.3.7 Exemplos.

Para iniciar os exemplos utilizaremos duas “funções” básicas: `Integer.parseInt (String umaString)` e `Float.parseFloat(String umaString)`. Estas duas “funções”, a primeira de

inteiro e a segunda de números decimais possuem como parâmetro uma String e como resultado a transformação da referida String em Inteiro ou em um número decimal.

Para executar os exemplos é necessário também saber o que significa o comando: `public static void main (String args[]);` . Este comando estará associado a todos os programas mostrados nos exemplos. Este comando significa que quando se executa o programa/classe é a função mais que é levada a cabo. Esta função aceita como parâmetro uma variável que é definida com o `args` e que é um vetor de Strings. No caso em dos exemplos os valores serão passados através deste parâmetro. Após compilar este programa/classe deve-se da linha de comando em DOS executar a seguinte instrução: `java Maior 1 2 <enter>`.

Veja os exemplos abaixo:

```
public class Maior { //1
    public static void main(String args[ ]) { //2
        int umInteiro; //3
        int outroInteiro; //4
        umInteiro = Integer.parseInt(args[0]); //5
        outroInteiro = Integer.parseInt(args[1]); //6
        if (umInteiro > outroInteiro) { //7
            System.out.println(" o maior é " + umInteiro); //8
        } //9
        else System.out.println(" o maior é " + outroInteiro); //10
    } //11
} //12
```

Código 1.15 - exemplo MaiorNúmero.

No exemplo acima repare o a declaração da função do programa/classe na linha 2. Na linha 3 e 4 são declarados as variáveis inteiras que serão compradas. Nas linhas cinco e seis são os valores de `args` são convertidos para inteiro. Por fim na linha sete os valores são comprados e é mostrado o maior.

```
public class CalculadoradeInteiros { //1
// esta calculadora opera com numeros inteiros sendo definidas apenas
// tres operações. Para soma utilize +; para subtração utilize -; e
// para multiplicação
// utilize qualquer outro caracter ou o carcater "*" (com dupla
// aspas).
    public static void main(String args[ ]) { //2
        int umInteiro; //3
        int outroInteiro; //4
        int resultado; //5
```

```

char operador; //6
umInteiro = Integer.parseInt(args[0]); //7
operador = args[1].charAt(0); //8
outroInteiro = Integer.parseInt(args[2]); //9
if (operador == '+') { //10
    resultado = umInteiro + outroInteiro; //11
} //12
else if (operador == '-') { //13
    resultado = umInteiro - outroInteiro; //14
} //15
else { //16
    resultado = umInteiro * outroInteiro; //17
} //18
System.out.println ( " o resultado é " + resultado); //19
} //20
} //21

```

Código 1.16 - exemplo calculadora.

No código acima verifica-se que são os parâmetros passados. O primeiro é o primeiro operando, o segundo é o operador e o terceiro é o segundo operando. O primeiro operando é obtido através da aplicação da função `parseInt` na linha sete e nove. A linha oito obtém o operador através da aplicação da função `charAt (0)`. As linhas 10, 13 e 16 testam o tipo de operação. A linha 19 mostra o resultado.

1.5 A plataforma Java.

Atualmente Java possui três plataformas de desenvolvimento. J2ME (Java 2 Platform, Micro Edition), Java 2 Platform, Standard Edition (J2SE) e J2EE (Java 2 Platform, Enterprise Edition).

A plataforma J2ME é a plataforma para dispositivos tais como telefones, palm etc. Em geral esta plataforma se presta ao desenvolvimento de aplicações com limites de recursos e de memória de computador.

A plataforma J2SE é a plataforma padrão de desenvolvimento, incluindo capacidade para o desenvolvimento de applets e Java beans. Esta plataforma se aplica ao desenvolvimento de sistemas centrados em rede e desktop. Esta plataforma possui dois produtos: o Java 2 SDK, Standard Edition e o Java 2 Runtime Environment, Standard Edition. O primeiro é um conjunto de principal de API da linguagem de programação Java.

A plataforma J2EE é a plataforma para o desenvolvimento de aplicações comerciais em larga escala. Neste caso a plataforma se aplica a sistemas de missão crítica e de conectividade com a web. A plataforma base da J2EE é a J2SE.

Parte Um – Conceitos Iniciais.

Capítulo 2 - Classes e Objetos

2.1. Conceitos de Classes e de Objetos.

Orientação a Objetos é uma forma de interpretar, modelar e entender o mundo sob a perspectiva de objetos. Assim, dois conceitos são importantes: o de Classes e o de Objetos. O entendimento destes conceitos é vital para todo o aprendizado Java e de orientação a objetos, portanto fique atento! Uma boa introdução a Classes e Objetos pode ser encontrada em Deitel[2005] e Larman[2004].

Objeto é um conceito básico quando se trata deste tema, já que é o núcleo de todos os outros conceitos. Em termos práticos, tudo a nosso redor pode ser considerado objeto: papel, lápis, computador, carro, estante etc. Objetos não se resumem a objetos visíveis ou concretos, eles podem ser também objetos abstratos: amizade, saldo, felicidade, amor etc.

Dois conceitos estão associados a objetos: eles possuem estados, expressos em variáveis/atributos, e comportamentos, que expressam funcionalidades/métodos/operações. Por exemplo: um lápis possui como estado a cor, o tipo, o comprimento. Possui também comportamento tais como: escrever e apagar. Assim, as variáveis de um objeto guardam o estado de um objeto e os métodos se aplicam as ações que modificam os estados do objeto.

Desta forma quando se faz referência a um objeto consideram-se estes dois componentes de forma integrada. Esta perspectiva se diferencia da forma tradicional de perceber o mundo como um conjunto de funcionalidades e um conjunto de dados de forma meramente interrelacionada.

Logo, uma possível definição de objetos pode ser: *Um objeto é um artefato de software que é composto por um conjunto de variáveis e os respectivos métodos.*

A ilustração da FIGURA 1 mostra um exemplo de objetos.

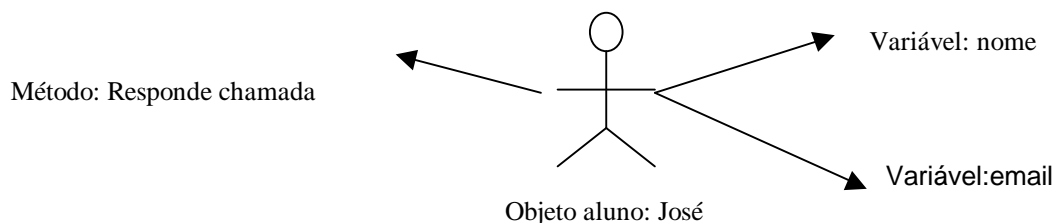


Figura 2.1. Um objeto

Uma das características da Figura 1 é que os métodos do objeto envolvem as variáveis do mesmo. De maneira geral, os valores das variáveis do objeto somente podem ser obtidos via um método específico.

Diz-se que as variáveis, neste caso, estão encapsuladas pelos métodos, ou seja, estas variáveis não estão visíveis a outros objetos para acessá-las é necessário a utilização de um método.

No mundo a nossa volta presenciamos vários objetos de um mesmo tipo. Por exemplo, sabemos qual o conceito de lápis. Sempre que alguém nos fala de um lápis sabemos qual o conceito de que se está falando, suas principais variáveis e seu comportamento típico.

Desta forma, quando nos referenciamos a conceitos, e não a objetos, estamos tratando de classes. Logo, uma definição para classe é: *uma classe é uma referência, um conceito que estabelece quais os métodos e quais as variáveis são comuns a todos os objetos desta classe.*

Se classes são conceitos e objetos existem na realidade então é necessário criar objetos a partir da classe. O processo de criação de um objeto a partir da classe é denominado instanciação. Assim todos os objetos que existem em um determinado momento foram, de alguma forma instanciados a partir de uma classe. Não existe objeto sem uma classe referência. Em orientação a objetos os objetos só existem se estes foram criados em um processo de instanciação de uma determinada classe.

2.2. Em Java.

Java é uma linguagem totalmente Orientada a Objetos. Esta linguagem possui tratamentos específicos para a definição de classes e para a criação de objetos. Lembre-se que tanto objetos e classes são compostos de variáveis e de métodos, logo na definição de uma classe o que será feito é a declaração destas respectivas variáveis e métodos. Estas definições de variáveis e de métodos, dependendo de como são feitas, podem pertencer tanto a ao objeto instanciado quanto a classe a que se referencia.

Considere o exemplo abaixo. Antes de digita-lo deve-se ter em mente que o nome do arquivo que contém a classe deve ter o mesmo nome da classe, assim o arquivo que contém o código abaixo possui o nome `Aluno.java`. O “//” é uma forma de documentação de código e os números “1” e “2” servem apenas para referenciar a linha do código Java.

```
public class Aluno {           //1
}                               //2
```

Código 2.1 - Classe Aluno - inicial.

Este pedaço de código (1.1) é uma definição inicial da classe Aluno. Entretanto, esta classe não representa de forma precisa um conceito ao qual estamos acostumados, é necessário criar variáveis para esta classe.

```
public class Aluno {           //1
    // variáveis                //2
    public String primeiroNome; //3
```

```

public String nomeFamilia;      //4
public int idade;               //5
public char sexo;               //5
}                                //6

```

Código 2.2 - Classe Aluno - com variáveis.

Este novo código (1.2) adiciona, à classe aluno, variáveis relativas ao aluno tais o seu primeiro nome, seu nome de família, sua idade e seu sexo. Falta a definição de uma funcionalidade para esta classe, é o que será definido no código 1.3.

```

public class Aluno {           //1
    // variáveis                2
    public String primeiroNome; //3
    public String nomeFamilia;  //4
    public int idade;           //5
    public char sexo;           //6
    // métodos                  7
    public Aluno(String primeiroNome, String nomeFamilia, //8
        int idade, char sexo ) { //9
        this.primeiroNome = primeiroNome; //10
        this.nomeFamilia = nomeFamilia; //11
        this.idade = idade; //12
        this.sexo = sexo; //13
    } //14
} //15

```

Código 2.3 - Classe Aluno - com variáveis e métodos

Pronto! O código 1.3 possui uma definição de uma classe e as respectivas declarações das variáveis e de seus métodos. Uma classe pode ter conceitualmente um conjunto qualquer de variáveis e de métodos. Vale lembrar que o código 1.3 é uma definição de uma classe, logo este é apenas um conceito daquilo que se quer representar. O que está definido vale para qualquer aluno, mas NÃO é um aluno em particular.

Para se criar um objeto é necessário instanciar a classe, quantas vezes for necessário. Assim, se desejarmos criar três alunos será necessário realizar três instanciações. Em particular o método Aluno possui o mesmo nome da classe. Este método é especial pois é denominado o *construtor* da classe, ou seja, é aquele responsável pela criação correta do objeto de uma determinada classe.

Para criar o aluno José, João e Maria será definida uma classe de demonstração que criará estes objetos, conforme apresentado no código 1.4.

```
public class ExemploAluno { //1
    public static void main (String[] args){ //2
        Aluno alunoJose = new Aluno ("José", "Campos", 37, 'm'); //3
        Aluno alunoJoao = new Aluno ("João", "Campos", 20, 'm'); //4
        Aluno alunaMaria= new Aluno ("Maria", "Campos", 18, 'f');//5
    } //6
} //7
```

Código 2.4 - Classe Aluno

Desta forma tem-se três objetos: alunoJose, alunoJoao e alunaMaria. Estes objetos são do tipo aluno, conforme mostrado no início das linhas três, quatro e cinco. O comando NEW, nestas mesmas linhas é o que invoca o método Aluno da classe Aluno e que faz criar os respectivos objetos com os valores passados por parâmetro.

Repare a definição do método main acima “public static void main (String[] args)”. Este método é um método da classe, conforme será explicado mais adiante.

2.4. Exercícios.

1. Especifique em Java uma classe Professor com os seguintes atributos: nome com tipo String, nomeFamilia com o tipo String, titulação com tipo String, anosdeAcademia do tipo int. Crie também uma classe chamada ExemploProfessor para instanciar 2 professores quaisquer.

Capítulo 3 - Variáveis e Métodos de Objetos e de Classes.

3.1. Conceitos de Variáveis e Métodos de um objeto e de uma Classe.

O objetivo desta aula é apresentar a motivação para se especificar variável e método tanto de uma classe quanto de um objeto.

No exemplo dado na aula anterior, da classe Aluno, todas as variáveis e métodos definidos para ela estavam associados ao objeto quando este era instanciado. Desta forma pode-se referenciar o valor de uma variável através da notação `umAluno.primeiroNome`, onde `umAluno` é uma instância de uma classe Aluno e `primeiroNome` uma variável associada a este objeto.

Poder-se-ia, da mesma forma, alterar o valor da variável do objeto, por exemplo: `umAluno.primeiroNome = "Christopher"`.

Entretanto existe a necessidade de guardar e manipular valores no âmbito da classe. Por exemplo, o que poderá ser feito caso se desejasse saber quantos objetos da classe Alunos foram criados? Se um programador adicionasse a variável `numAluno`, da mesma forma como se fez com as outras variáveis, cada vez que um aluno fosse instanciado esta variável estaria associada a este objeto. O que se deseja é ter o controle do número de alunos criados e que este valor esteja associado a própria classe Aluno. Desta forma a variável ao invés de pertencer ao objeto esta passa a pertencer à classe.

Assim para diferenciar as variáveis/métodos do objeto das variáveis e métodos da classe utiliza-se o comando **static** antes da declaração da variável ou do método.

Pode-se acessar a variável de classe através da própria classe ou das instâncias criadas por ela. Pode-se acessar o método de uma classe através da própria classe. Os métodos de classe não podem acessar a instancias criadas pela classe.

3.2. Em Java.

Considere o exemplo abaixo:

```
public class Aluno { //1
    // variáveis da classe //2
    public static int numAlunos = 0; //3
    // variáveis //4
    public String primeiroNome; //5
    public String nomeFamilia; //6
    public int idade; //7
    public char sexo; //8
    // métodos //9
    public Aluno(String primeiroNome, String nomeFamilia, //10
```

```

        int idade, char sexo ) { //11
        this.primeiroNome = primeiroNome; //12
        this.nomeFamilia = nomeFamilia; //13
        this.idade = idade; //14
        this.sexo = sexo; //15
    } //16
} //17

```

Código 3.1 - Classe Aluno com variável estática

As linhas dois e três apresentam uma pequena mudança em relação ao código 1.4. O código “public static int numAlunos = 0;” define uma variável que irá pertencer à classe e não ao objeto. O código abaixo cria as instancias do aluno e totaliza o numero de alunos criado na variável numAlunos.

```

Public class ExemploAlunoVariávelEstatica //1
    public static void main (String[] args){ //2
        Aluno umAluno = new Aluno ("Marcio", "Campos", 27, 'm'); //3
        Aluno.numAlunos = Aluno.numAlunos + 1; //4
        System.out.println("Criado Professor Marcio"); //5
        System.out.println("nome: " + umAluno.primeiroNome); //6
        System.out.println("nome familia: " + umAluno.nomeFamilia); //7
        System.out.println("idade: " + umAluno.idade); //8
        System.out.println("Sexo: " + umAluno.sexo); //9
        //10
        Aluno outroAluno = new Aluno ("José", "Xexeo", 35, 'm'); //11
        Aluno.numAlunos = Aluno.numAlunos + 1; //12
        System.out.println("Criado Professor Xexeo"); //13
        System.out.println("nome: " + outroAluno.primeiroNome); //14
        System.out.println("nome familia:" + outroAluno.nomeFamilia); //15
        System.out.println("idade: " + outroAluno.idade); //16
        System.out.println("Sexo: " + outroAluno.sexo); //17
        System.out.println("numero de alunos" + Aluno.numAlunos); //18
    } //19
} //20

```

Código 3.2 - classe demonstração da variável estática.

Como se pode notar na linha quatro e na linha 12 o código “Aluno.numAlunos = Aluno.numAlunos + 1;” realiza a adição de cada objeto criado a variável de classe numAlunos.

Desta forma a classe Aluno possuirá uma variável definida como numAlunos que guardará o número de instancias criadas para esta classe.

De forma semelhante o método definido na linha dois, “public **static** void main (String[] args){”, possui a palavra reservada **static** o que determina que o método sendo definido é um método da classe. Esse método **main**, em particular, é executado sempre que se executa a classe.

Na linha três e onze o construtor Aluno da classe Aluno foi utilizado para instanciar cada objeto. As linhas seis a nove e 14 a 17 foram definidas para mostrar o objeto criado.

O código acima poderia ser melhorado utilizando-se mais apropriadamente os conceitos de variáveis e métodos de objetos e de classes. Por exemplo, as linhas 14 a 17 mostram o objeto, logo esta funcionalidade poderia ser um método do objeto já está associada ao próprio objeto. Assim poderíamos adicionar o seguinte método ao código 2.1:

```
public void mostraAluno() { //1
    System.out.println("nome: " + this.primeiroNome); //2
    System.out.println("nome familia: " + this.nomeFamilia); //3
    System.out.println("idade: " + this.idade); //4
    System.out.println("Sexo: " + this.sexo); //5
} //6
```

Código 3.3 - Método MostraObjeto.

O código 2.3 mostra na linha um a declaração do nome do método, a seguir nas linhas dois a cinco, mostra o objeto sendo mostrado. Repare a referência ao objeto corrente “this” seguida da variável do objeto. Esta referência faz com que o objeto que está sendo manipulado mostre seus próprios valores contidos em suas variáveis.

De forma semelhante, olhando para o código 2.1 verificamos que ao se adicionar “1” a variável numAlunos que esta funcionalidade simples está associada a uma operação da classe, logo poderíamos criar um método da classe para implementar esta funcionalidade, como mostra o código 2.4.

```
public static void contaAlunos() { //1
    numAlunos = numAlunos + 1; //2
} //3
```

Código 3.4 - Método da classe contaAlunos.

Repare a palavra static, na linha um, que define que o método é da classe. De forma semelhante poderíamos criar um método da classe que mostrasse o total de alunos criados, conforme o código 2.5

```
public static void mostraTotalAlunos () { //1
    System.out.println ("o total de alunos é " + numAlunos); //2
} //3
```

Código 3.5 - Método da classe mostraTotalAlunos.

Note que na linha dois não foi utilizado o a palavra “this” junto a “numAlunos” já que não existe um objeto associado pois esta é uma variável da classe.

O código final da classe Aluno, ao final destas alterações ficaria como o do código 2.6

```
public class Aluno { //1
    // variáveis da classe //2
    public static int numAlunos = 0; //3
    //métodos da classe //4
    public static void contaAlunos() { //5
        numAlunos = numAlunos + 1; //6
    } //7
    public static void mostraTotalAlunos () { //8
        System.out.println("o total de alunos é "+numAlunos); //9
    } //10
    // variáveis do objeto //11
    public String primeiroNome; //12
    public String nomeFamilia; //13
    public int idade; //14
    public char sexo; //15
    // métodos do objeto //16
    public Aluno(String primeiroNome, String nomeFamilia, //17
        int idade, char sexo ) { //18
        this.primeiroNome = primeiroNome; //19
        this.nomeFamilia = nomeFamilia; //20
        this.idade = idade; //21
        this.sexo = sexo; //22
    } //23
    public void mostraAluno() { //24
        System.out.println("nome: " + this.primeiroNome); //25
        System.out.println("nome familia:"+this.nomeFamilia); //26
        System.out.println("idade: " + this.idade); //27
        System.out.println("Sexo: " + this.sexo); //28
    } //29
} //30
```

Código 3.6 - Classe Alunos final.

A classe ExemploAlunoVariavelEstatica ficaria conforme o código 2.7.

```
public class ExemploAlunoVariávelEstatica //1
    public static void main (String[] args){ //2
        Aluno umAluno = new Aluno ("Marcio", "Campos", 27, 'm'); //3
        Aluno.contaAlunos(); //4
        System.out.println("Criado Professor Marcio"); //5
        umAluno.mostraAluno(); //6
        //7
        Aluno outroAluno = new Aluno ("José", "Xexeo", 35, 'm'); //8
        Aluno.contaAlunos(); //9
        System.out.println("Criado Professor Xexeo"); //10
        outroAluno.mostraAluno(); //11
        Aluno.mostraTotalAlunos(); //12
    } //13
} //14
```

Código 3.7 - Classe DemoAlunoVariávelEstatica.

3.4. Exercícios.

1. Repita para a classe professor, criada no exercício do capítulo anterior, a mesma estrutura de código desenvolvida para a classe aluno neste capítulo.

Capítulo 4 - Mensagens e Visibilidade.

4.1. Conceitos de mensagens e visibilidade.

O objetivo desta aula é apresentar como os conceitos de mensagens, visibilidade e de escopo influenciam a definição e o uso dos objetos e das classes e das variáveis e dos métodos.

Os objetos não vivem sozinhos no mundo. Assim que se cria um objeto este possui propriedades e métodos, conforme visto anteriormente. Esses métodos são invocados, chamados, ou pelo próprio objeto ou por outro objeto. Quando esta interação entre objetos ocorre diz-se que os objetos estão trocando mensagens.

Entretanto nem todas as mensagens podem ser chamadas. Podem existir métodos e atributos que não possam ser executados diretamente a partir de outros objetos devido a sua visibilidade.

No caso a seguir, a classe ExemploMensagemVisibilidade faz uma chamada ao método de criação de um objeto Aluno.

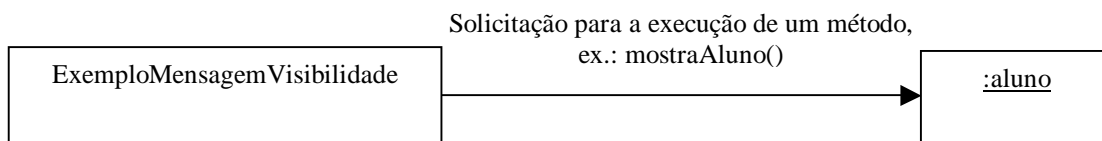


Figura 4.1 mensagens.

No caso da figura 3.1 o método chamado pode ser válido ou não. Isso dependerá da visibilidade do método declarada para a classe Aluno. O mesmo conceito pode se aplicar a uma variável, caso a classe ExemploMensagemVisibilidade, ou um de seus objetos, quisesse acessar o conteúdo de uma variável de Aluno, ou de um de seus objetos, o resultado desta operação dependeria da visibilidade desta variável na classe aluno.

4.2. Em Java.

Primeiro será tratado o conceito de visibilidade. Abaixo são listadas as duas classes: a classe Aluno e a classe ExemploMensagemVisibilidade. Os atributos básicos de visibilidade são: public, private. O atributo public permite acesso direto ao método ou a variável e o atributo private não permite o acesso a partir de outro objeto. Em Java, quando uma determinada classe ou objeto tenta acessar um método ou variável que tenha visibilidade private ocorre um erro de compilação. Execute-as e veja o que acontece.

```
public class Aluno { //1
    // variáveis da classe 2
```

```

    public static int numAlunos = 0; //3
// métodos da classe //4
    public static void contaAlunos() { //5
        numAlunos = numAlunos + 1; //6
    } //7
    public static void mostraTotalAlunos () { //8
        System.out.println("o total de alunos é "+numAlunos); //9
    } //10
// variáveis do objeto //11
private String primeiroNome; //12
private String nomeFamilia; //13
private int idade; //14
private char sexo; //15
// métodos do objeto //16
public Aluno(String primeiroNome, String nomeFamilia, //17
    int idade, char sexo ) { //18
    this.primeiroNome = primeiroNome; //19
    this.nomeFamilia = nomeFamilia; //20
    this.idade = idade; //21
    this.sexo = sexo; //22
} //23
public void mostraAluno() { //24
    System.out.println("nome: " + this.primeiroNome); //25
    System.out.println("nome familia:"+this.nomeFamilia); //26
    System.out.println("idade: " + this.idade); //27
    System.out.println("Sexo: " + this.sexo); //28
} //29
} //30

```

Código 4.1 - Classe aluno com variáveis de objetos private.

```

public class ExemploMensagemVisibilidade //1
    public static void main (String[] args){ //2
        Aluno umAluno = new Aluno ("Marcio", "Campos", 27, 'm'); //3
        contaAlunos(); //4
        System.out.println("Criado Professor Marcio"); //5
        umAluno.idade = 22; //6
    } //7
} //8

```

Código 4.2 - Classe DemoMensagemVisibilidade.

No exemplo acima, no código 3.2, a linha seis ocorreria um erro de compilação. Por que? Porque a classe DemoMensagemVisibilidade tenta alterar a idade do aluno umAluno e esta variável é inacessível a outras classes, apenas os próprios método do objeto umAluno podem manipulá-la. Desta forma caso se deseje fazer alterações sobre os dados de aluno deve-se criar um método na classe Aluno para que os objetos instanciados por esta classe possam alterar os seus dados. O código 3.3 adiciona o método alterarAluno() para lidar com a nova visibilidade das variáveis dos objetos de aluno.

```
public class Aluno { //1
    // variáveis da classe //2
    public static int numAlunos = 0; //3
    //métodos da classe //4
    public static void contaAlunos() { //5
        numAlunos = numAlunos + 1; //6
    } //7
    public static void mostraTotalAlunos () { //8
        System.out.println("o total de alunos é "+numAlunos); //9
    } //10
    // variáveis do objeto //11
    private String primeiroNome; //12
    private String nomeFamilia; //13
    private int idade; //14
    private char sexo; //15
    // métodos do objeto //16
    public Aluno(String primeiroNome, String nomeFamilia, //17
        int idade, char sexo ) { //18
        this.primeiroNome = primeiroNome; //19
        this.nomeFamilia = nomeFamilia; //20
        this.idade = idade; //21
        this.sexo = sexo; //22
    } //23
    public void mostraAluno() { //24
        System.out.println("nome: "+this.primeiroNome); //25
        System.out.println("nome familia:"+this.nomeFamilia); //26
        System.out.println("idade: " + this.idade); //27
        System.out.println("Sexo: " + this.sexo); //28
    }
}
```

```

    } //29
    public void alterarDados (String primeiroNome, //30
        String nomeFamilia, int idade, char sexo ) { //31
        if (primeiroNome != this.primeiroNome) //32
            this.primeiroNome = primeiroNome; //33
        if (nomeFamilia != this.nomeFamilia) //34
            this.nomeFamilia = nomeFamilia; //35
        if (idade != this.idade) //36
            this.idade = idade; //37
        } //38
    } //39
} //40

```

Código 4.3 - Classe aluno com método alterarDados.

Desta forma, de acordo com o código 3.3, a classe DemoMensagemVisibilidade deve ser alterada conforme o código 3.4.

```

public class ExemploMensagemVisibilidade { //1
    public static void main (String[] args){ //2
        Aluno umAluno = new Aluno ("Marcio", "Campos", 27, 'm'); //3
        Aluno.contaAlunos(); //4
        System.out.println("Criado Professor Marcio"); //5
        umAluno.alterarDados("Marcio", "Campos", 22, 'm'); //6
    } //7
} //8

```

Código 4.4 - Classe ExemploMensagemVisibilidade.

Se quisermos ter acesso as variáveis dos objetos de forma a manipula-los devemos criar métodos que acessem essas variáveis e que retornem os valores das mesmas, por exemplo:

```

public int valor_idade() { //1
    return this.idade //2
} //3

```

Código 4.5 - Método de retorno de valores de variáveis privadas.

Com este e outros métodos semelhantes para todas as variáveis podemos recuperar os valores destas variáveis e operá-las como desejarmos.

Se adicionarmos o método acima para a classe aluno poderíamos reescrever a classe ExemploMensagemVisibilidade para recuperar o valor alterado e mostrá-lo, conforme exemplificado no código 3.6.

```
public class ExemploMensagemVisibilidade //1
    public static void main (String[] args){ //2
        Aluno umAluno = new Aluno ("Marcio", "Campos", 27, 'm'); //3
        Aluno.contaAlunos(); //4
        System.out.println("Criado Professor Marcio"); //5
        umAluno.alterarDados("Marcio", "Campos", 22, 'm'); //6
        int idade_alterada = umAluno.valor_idade(); //7
        System.out.println("Idade alterada é:"+idade_alterada); //8
    } //9
} //10
```

Código 4.6 - Classe DemoMensagemVisibilidade.

Aos métodos também podem ser aplicados os conceitos de visibilidade. Quando um método é private indica que apenas outros métodos definidos para o objeto podem acessá-lo, assim este será um método exclusivo dos outros métodos da classe.

A classe DemoMensagemVisibilidade nos dá um exemplo de troca de mensagens entre a classe DemoMensagemVisibilidade e o objeto umAluno instanciado da classe Aluno.

4.4 Exercícios.

1. Especifique em Java uma classe Aluno os outros métodos de recuperação de valores de cada variável de objeto.

Capítulo 5 - Sobrecarga.

5.1. Introdução.

Construtores são utilizados para instanciar objetos de uma classe. Já vimos como os construtores são utilizados nos exemplos anteriores. Como por exemplo, o método Aluno definido na aula três. Entretanto um método pode ser utilizado de várias formas. Estas várias formas de invocação, de um mesmo método, chama-se sobrecarga.

Os construtores são úteis para que possamos inicializar campos de um objeto. Desta forma garante-se que o objeto criado segue um determinado número de regras definidas para aquele construtor. Em realidade, para se criar objetos não é necessário de um construtor, mas sua utilização garante maior consistência e manutenibilidade ao código. Já utilizamos construtores antes e sabemos como eles se comportam. A utilização dos construtores garante que a criação dos objetos seguirá uma funcionalidade padrão.

Um método é identificado como um construtor quando o nome do método é igual ao da classe. Já foi visto também a palavra-chave **new** é utilizada em conjunto com os construtores para instanciar os objetos.

Uma das características específicas na definição de métodos é a capacidade de se definir vários métodos com diferentes assinaturas. Uma assinatura é definida pelo nome do método e os correspondentes tipos dos parâmetros deste.

Desta forma poderíamos criar/instanciar um Aluno com diferentes tipos de assinaturas. Esse mecanismo se chama sobrecarga.

A sobrecarga não é exclusivo de construtores esta pode ser utilizada em métodos de maneira geral.

5.2. Em Java.

Considere o exemplo do código 4.1.

```
public class Aluno { //1
    // variáveis da classe //2
    public static int numAlunos = 0; //3
    //métodos da classe //4
    public static void contaAlunos() { //5
        numAlunos = numAlunos + 1; //6
    } //7
    public static void mostraTotalAlunos () { //8
        System.out.println ("o total de alunos é " + numAlunos);
    } //9
} //10
```

```

// variáveis do objeto      11
private String primeiroNome; //12
private String nomeFamilia; //13
private int idade;          //14
private char sexo;         //15
// métodos do objeto      16
public Aluno() {           //17
    this.primeiroNome = "Teste"; //18
    this.nomeFamilia = "Default"; //19
    this.idade = 18;       //20
    this.sexo = 'm';      //21
}                           //22

public Aluno(String primeiroNome, String nomeFamilia, //23
    int idade, char sexo ) { //24
    this.primeiroNome = primeiroNome; //25
    this.nomeFamilia = nomeFamilia; //26
    this.idade = idade; //27
    this.sexo = sexo; //28
} //29
public void mostraAluno() { //30
    System.out.println("nome: " + this.primeiroNome); //31
    System.out.println("nomefamilia: "+this.nomeFamilia); //32
    System.out.println("idade: " + this.idade); //33
    System.out.println("Sexo: " + this.sexo); //34
} //35
public void alterarDados (String primeiroNome, //36
    String nomeFamilia, int idade, char sexo ) { //37
    if (primeiroNome != this.primeiroNome) //38
        {this.primeiroNome = primeiroNome}; //39
    if (nomeFamilia != this.nomeFamilia) //40
        {this.nomeFamilia = nomeFamilia}; //41
    if (idade != this.idade) //42
        { this.idade = idade}; //43
    } //44
} //45
} //46

```

Código 5.1 - Classe aluno com sobrecarga do construtor.

Repare que no código acima temos dois construtores. No primeiro método é criado um aluno padrão, onde não é passado nenhum parâmetro. No segundo é criado um aluno passando todos os parâmetros. Caso uma outra classe invocasse o construtor da classe aluno poderia, dependendo da necessidade instanciar um aluno teste ou meramente um aluno real. O código 4.2 mostra a classe que utiliza estes dois construtores.

```
public class ExemploSobrecarga //1
    public static void main (String[] args){ //2
        Aluno umAluno = new Aluno ("Marcio", "Campos", 27, 'm'); //3
        contaAlunos(); //4
        Aluno outroAluno = new Aluno (); //5
        ContaAlunos(); //6
    } //7
} //8
```

Código 5.2 - Classe DemoSobrecarga.

5.4 Exercícios.

1. Crie um outro método construtor para criar alunos de intercâmbio onde a faculdade de origem é um parâmetro importante.
2. Crie um método mostraAluno onde seja passado como parâmetro o número de linhas que deve ser puladas para se imprimir os detalhes dos alunos.

Capítulo 6 – Delegação e Composição.

6.1 Introdução.

Um dos aspectos inerentes a orientação a objetos é a capacidade de reaproveitar código ou uma estrutura anteriormente definida. Esta vantagem é importante quando um dos fatores que mais pesam no custo de software é o de manutenção. Neste capítulo será visto conceito de delegação, uma das formas de reaproveitamento de código.

Na delegação, uma classe base é definida e esta é referenciada por outra classe. Por exemplo, um objeto da classe Computador pode possuir entre seus atributos uma referência um objeto do tipo monitor. Assim, a criação de um objeto computador faz uso dos métodos na classe monitor.

Desta maneira, a delegação permite a reutilização, ou composição, de classes já existentes como instâncias de novas classes. Sob este aspecto as classes originais ficam contidas nas novas classes.

6.2. Em Java.

Considere os código 5.1 a 5.5: teclado, monitor, mouse e gabinete. Podemos criar estes objetos individualmente, ou seja, criar uma classe para teclado, monitor, mouse e gabinete.

```
public class Teclado { //1
    // variáveis de objeto //2
    private String numero_serie; //3
    private String formato; //4
    // construtor //5
    public Teclado (String numero_serie, String formato) { //6
        this.numero_serie = numero_serie; //7
        this.formato = formato; //8
    } //9
} //10
```

Código 6.1 - Classe Teclado

```
public class Gabinete { //1
    // variáveis de objeto //2
    private String patrimonio; //3
    private String formato; //4
    // construtor //5
    public Gabinete (String patrimonio, String formato) { //6
        this.patrimonio = patrimonio; //7
        this.formato = formato; //8
    } //9
} //10
```

```

    } //9
} //10

```

Código 6.2 - Classe Gabinete

```

public class Mouse { //1
    // variáveis de objeto //2
    private String numero_serie; //3
    private String tecnologia; //4
    // construtor //5
    public Mouse (String numero_serie, String tecnologia) { //6
        this.numero_serie = numero_serie; //7
        this.tecnologia = tecnologia; //8
    } //9
} //10

```

Código 6.3 - Classe Mouse

```

public class Monitor { //1
    // variáveis de objeto //2
    private String numero_serie; //3
    private String fabricante; //4
    // construtor //5
    public Monitor (String numero_serie, String fabricante) { //6
        this.numero_serie = numero_serie; //7
        this.fabricante = fabricante; //8
    } //9
} //10

```

Código 6.4 - Classe Monitor

```

public class Computador { //1
    // variáveis de objeto //2
    private String sala; //3
    private String usuario; //4
    private int numero_computador; //5
    private Teclado umTeclado; //6
    private Gabinete umGabinete; //7
    private Mouse umMouse; //8

```

```

private Monitor umMonitor; //9
// construtor //10
public Computador (String sala, String usuario, int //11
numero_computador, Teclado umTeclado, //12
Gabinete umGabinete, Mouse umMouse, Monitor umMonitor){ //13
    this.sala = sala; //14
    this.usuario = usuario; //15
    this.numero_computador = numero_computador; //16
    this.umTeclado = umTeclado; //17
    this.umGabinete = umGabinete; //18
    this.umMouse = umMouse; //19
    this.umMonitor = umMonitor; //20
} //21
} //22

```

Código 6.5 - Classe Computador

No código 5.1 a 5.4 foram criadas classes que em sua essência não dependem de outras classes para sua existência. Entretanto, a classe Computador é na realidade uma composição de outras classes. Essa abordagem de criação de objetos maiores a partir de objetos menores possibilita a existência de independente dos objetos menores com seus próprios métodos e atributos.

A classe ExemploDelegaçãoComposição, do código 5.6 exemplifica o uso de como a partir da criação de objetos menores, delegando a cada um deles responsabilidades inerentes aos mesmos, pode-se criar objetos maiores através da composição.

```

public class ExemploDelegacaoComposicao { //1
    // método da classe //2
    public static void main (String args[]) { //3
        Teclado umTeclado = new Teclado("12345SN", "ABNT"); //4
        Gabinete umGabinete = new Gabinete("PTR0725", "ATX"); //5
        Mouse umMouse = new Mouse("4567SN", "infravermelho"); //6
        Monitor umMonitor = new Monitor("1011SN", "DELL"); //7
        Computador umComputador = new Computador("217", "Jo", //8
15, umTeclado, umGabinete, umMouse, umMonitor); //9
    } //10
} //11

```

Código 6.6 - Classe ExemploDelegacaoComposicao

Repare no código 5.6 que foi necessário a criação dos objetos teclado, gabinete, mouse e monitor para que fosse possível criar o objeto computador.

6.4.Exercício.

Construa classes que especifique o conceito de uma casa.

Capítulo 7 - Herança.

7.1. Introdução.

Herança é uma outra forma de reutilizar código. Diferentemente da delegação, que possibilita a criação de grandes objetos a partir de outros menores, a herança permite a criação de objetos que são parecidos, em comportamento, com outros objetos ancestrais. Na delegação um multiobjeto é composto de outros objetos ao passo que herança um objeto é criado a partir de outro com comportamento parecido.

O que é herança? Na herança uma classe é definida em termos de outras classes. Por exemplo, carros e motocicleta podem ser definidos em termos da classe veículos de locomoção. Neste caso carro e motocicletas são subclasses da classe veículos de locomoção. A classe veículos de locomoção é definida com uma superclasse.

As subclasses possuem, por definição, comportamentos e estados semelhantes aos da superclasse. De maneira geral, pode-se dizer que as subclasses herdam tanto as variáveis quanto os métodos da superclasse. Uma definição encontrada no Java Tutorial é: *uma subclasse é uma classe que estende outra classe sendo que esta subclasse herda os estados(atributos) e comportamento (métodos) de todas as classes ancestrais; o termo superclasse se refere a classe imediatamente ancestral assim como todas as outras classes ancestrais.*

Entretanto, a subclasse não está limitada as variáveis e aos métodos da superclasse. Para caracterizá-la melhor novas variáveis ou métodos podem ser adicionadas à subclasse, assim como podem ser suprimidos, ou até mesmo redefinidos.

De forma geral as seguintes regras se aplicam:

- As subclasses herdam variáveis e métodos cuja visibilidade seja pública ou protegida.
- As subclasses herdam da superclasse as variáveis e os métodos sem especificação de visibilidade desde que a classe esteja no mesmo pacote.
- As subclasses NÃO herdam membros da superclasse se a subclasse declarar uma variável ou método de mesmo nome. No caso de variáveis, as variáveis definidas na subclasse ocultam aquelas definidas nas superclasses; no caso dos métodos, os métodos definidos na subclasse redefinem aqueles definidos nas superclasses.

7.2 Em Java.

Java implementa o conceito de herança através da palavra Extends. No nosso exercício do aluno vamos aproveitar a classe Aluno definida em 4.1 e estendê-la através de herança para a classe AlunodeIntercâmbio. Verificamos inicialmente que um aluno de intercâmbio é também um aluno regular, logo podemos aproveitar a definição da classe aluno para a definição da nova classe. O código

da classe AlunodeIntercâmbio é mostrado no código 6.1. Nesta caso a classe Aluno é *superclasse* e a classe AlunodeIntercambio é uma *subclasse*.

```
public class AlunodeIntercambio extends Aluno{ //1
    // variáveis do objeto //2
    public String faculdade_origem; //3
    // métodos do objeto //4
    public AlunodeIntercambio (String nome, String familia, //5
        int idade, char sexo, String faculdade_origem){ //6
        super (nome, familia, idade, sexo); //7
        this.faculdade_origem = faculdade_origem; //8
    } //9
} //10
```

Código 7.1 - Classe AlunodeIntercambio.

Observe que a palavra super refere-se ao construtor da classe ancestral. Desta forma aproveita-se o código definido na classe ancestral. O código 6.2 demonstra o uso da herança.

```
public class ExemploHerança { //1
    public static void main (String args[]) { //2
        Aluno umAluno = new Aluno(); //3
        Aluno umAlunodeIntercambio = new AlunodeIntercambio("Jose", //4
            "Alves", 35, 'm', "UniverSidade"); //4
    } //5
} //6
```

Código 7.2 - Classe ExemploHerança.

Um método pode ser modificado pela classe herdeira. O código 4.1 possui o método mostraAluno. Poderíamos criar um outro método com o mesmo nome para mostrar o aluno de intercâmbio. Este método está descrito no código 6.3.

```
public void mostraAluno ( ) { //1
    super.mostraAluno(); //2
    System.out.println ("Origem" + this.faculdade_origem //3
} //4
```

Código 7.3 - Método mostraAluno da classe AlunodeIntercambio.

O código completo de aluno de intercâmbio ficaria, então, como mostrado na figura 6.4.

```

public class AlunodeIntercambio extends Aluno{ //1
    // variáveis do objeto //2
    public String faculdade_origem; //3
    // métodos do objeto //4
    public AlunodeIntercambio (String nome, String familia, //5
int idade, char sexo, String faculdade_origem){ //6
    super (nome, familia, idade, sexo, faculdade_origem); //7
    this.faculdade_origem = faculdade_origem; //8
} //9
public void mostraAluno ( ) { //10
    super.mostraAluno(); //11
    System.out.println ("Origem"+this.faculdade_origem) //12
} //13
} //14

```

Código 7.4 - a classe AlunodeIntercambio completa.

Em Java uma classe pode herdar ou estender apenas uma única classe.

Vale notar com este exemplo que agora se possui dois métodos com mesmo nome e assinatura: o da classe aluno e o da classe alunodeIntercâncio. Essa capacidade de chama-se polimorfismo, isto quer dizer o método mostraAluno() pois pode possuir várias formas, se comportar diferentemente, dependendo do objeto ao qual está associado. No caso da classe aluno este método mostra apenas as suas variáveis associadas ao objeto aluno e no caso da classe alunodeIntercambio o método mostra não apenas as suas variáveis como as de seu ancestral. Este conceito possibilita a extensibilidade das aplicações permitindo a definição de métodos que podem ser utilizados em toda um a hierarquia de classes em uma herança.

O código 6.5 mostra como o polimorfismo pode ser utilizado.

```

public class ExemploPolimorfismo { //1
    public static void main (String args[]) { //2
        Aluno umAluno; //3
        AlunodeIntercambio umAlunodeIntercambio = //4
New AlunodeIntercambio ("José", "Jack", 25, 'm', //5
"Minesota"); //6
        umAluno = umAlunodeIntercambio; //7
        umAluno.mostraAluno(); //8
    } //9

```

```
}
```

```
//10
```

Código 7.5 - a classe *ExemploPolimorfismo*

Ao executar o código acima verificar-se-á que apesar de umAluno ser um objeto da classe Aluno o método mostrarAluno() mostrará corretamente o aluno de intercâmbio.

Um outro desdobramento, quando se lida com hierarquias, é quanto a visibilidade das variáveis. Vimos, anteriormente, dois tipos de visibilidade public e private. No caso da classe AlunodeIntercambio, se este quisesse acessar as variáveis de Aluno ocorreria um erro de compilação pois as variáveis possuem visibilidade private.

Desta forma, quando a visibilidade é private, qualquer outra classe que queira acessar as variáveis de Aluno devem solicitar a um de seus métodos; inclusive as classes descendentes da classe Aluno.

Quando a visibilidade é pública o acesso é permitido indiscriminadamente à qualquer um.

Um meio do caminho entre a visibilidade public e o private é a visibilidade protected. Esta visibilidade permite que apenas as classes descendentes tenham acesso a classe base, em nosso exemplo a classe Aluno. Qualquer outra classe possui seu acesso negado.

7.4. Exercícios.

1. Aproveite a classe computador mostrada no código 5.5 e estenda-a para as classes ComputadorBásico com os atributos cdrom. Estenda a classe computador básico com o nome de ComputadorAvançado com os atributos dvd. Crie o método mostra que mostre o conteúdo do objeto.
2. Crie a classe ExemploComputadorHeranca para manipular os objetos das classes definidas.

Capítulo 8 – Classes Abstratas.

8.1. Introdução.

Classes abstratas são uma forma de se criar *subclasses* a partir de outras *superclasses* de modo a se reutilizar os métodos e os atributos desta última. Uma classe abstrata possui métodos abstratos, ou seja, não definidos. Desta forma por possuir um comportamento não totalmente especificado não se pode criar instâncias destas classes. Por sua vez, todas as outras classes que podem criar instâncias são *classes concretas*.

As *classes abstratas* são úteis quando se deseja trabalhar com conceitos, ou seja, define-se uma classe modelo e seus herdeiros é que se encarregam de implementar estes conceitos. Assim a *classe abstrata* serve como guia para a definição do comportamento dos herdeiros, das *subclasses*. Uma *subclasse* deve, então, obrigatoriamente, implementar todos os métodos abstratos, assim definidos na *superclasse*. Caso uma *subclasse* não implemente todos os métodos abstratos esta subclasse, por definição, será também abstrata.

8.2. Em Java.

Em Java define-se uma *classe abstrata* adicionando-se a palavra reservada `abstract` antes da classe ou da declaração de um método. Para uma classe herdar de uma classe abstrata basta utilizar o comendo `extends` da mesma forma que se descrevia uma herança. Considere o código 7.1 e 7.2.

```
public abstract class Contribuinte { //1
    // variáveis //2
    protected String nome; //3
    protected float total_receita; //4
    protected float total_descontos; //5
    // Métodos //6
    public Contribuinte (String nome, float total_receita, //7
        float total_descontos) { //8
        this.nome = nome; //9
        this.total_descontos = total_descontos; //10
        this.total_receita = total_receita; //11
    } //12
    public float rec_total_receita() { //13
        return total_receita; //14
    } //15
    public float rec_total_descontos() { //16
```

```

        return total_descontos; //17
    } //18
    // esse método é abstrato, pois não está definido. 19
    public abstract float imposto_a_pagar(); //20
} //21

```

Código 8.1 - Classe abstrata Contribuinte.

```

public class PessoaFisica extends Contribuinte { //1
    //variáveis 2
    protected String CPF; //3
    // Métodos 4
    public PessoaFisica (String nome, float total_receita, //5
float total_descontos, String CPF) { //6
        super(nome, total_receita, total_descontos); //7
        this.CPF = CPF; //8
    } //9
    public float imposto_a_pagar(){
        float total_a_pagar;
        return total_a_pagar = 0.25F * super.rec_total_receita();
    }
}

```

Código 8.2 - Classe Pessoa

O código 7.1 define uma classe contribuinte que é abstrata, pois possui um de seus métodos como abstratos. A classe PessoaFisica estende da classe Contribuinte, assim, herda os métodos e as variáveis. Entretanto a classe PessoaFisica necessitará implementar o método imposto_a_pagar() caso se deseje que ela se torne uma classe concreta. O código 7.2 demonstra a implementação do método tornando a referida classe em uma **classe concreta**.

O código 7.3 descreve a classe ExemploClasseAbstrata.

```

public class ExemploClasseAbstrata { //1
    public static void main (String args[]) { //2
        PessoaFisica umaPessoaFisica = //3
        New PessoaFisica("Jose",100.000F,30.000F, "123456-78");//4
        System.out.println(umaPessoaFisica.imposto_a_pagar());//5
        //repare que o código a seguir não funcionaria, pois
        // estaríamos instanciando uma classe abstrata.
    }
}

```

```
        // Contribuinte umaPessoaFisica =
        // New Contribuinte("Jose",100.000F,30.000F);
    }
}
```

Código 8.3 - Classe ExemploClasseAbstrata

8.3 Exercícios

1. Defina uma classe abstrata Cidadão, com um atributo documentodeIdentificacao e um método abstrato a ser implementado pelas classes Aluno e AlunodeIntercambio.

Capítulo 9 - Interfaces.

9.1. Introdução.

Interfaces podem ser definidas como um conjunto de protocolos que são implementados por uma ou mais classes. A classe que implementa a interface concorda com a implementação de todos os métodos definidos pela interface. Uma definição de interface é uma “coleção de definições de métodos, sem implementação, e de variáveis que são obrigatoriamente constantes”.

Devido ao fato de uma interface ser uma lista de métodos não implementados, e por sua vez abstratos, é necessário estabelecer uma diferença entre as interfaces e as classes abstratas. Logo,

- Uma interface não pode implementar nenhum método, por sua vez uma classe abstrata pode.
- Uma classe pode implementar várias interfaces, mas apenas uma superclasse abstrata.
- A interface não compõe a hierarquia da classe. Logo, classes não diretamente relacionadas podem implementar a mesma interface.

Um exemplo ilustra o conceito de interface. Imagine um usuário utilizando um controle remoto universal de uma televisão. Cabe ao usuário saber como operar apenas o controle remoto. Cada nova televisão pode vir com um design diferente, mas as únicas propriedades necessárias a operar a televisão são aquelas definidas pelo controle remoto, desde que a TV implemente as operações definidas pelo controle remoto.

Este tipo de abordagem possibilita a modularização. Caso a televisão quebre, basta comprar outra colocar no lugar da antiga que o conjunto de operações definidas no controle remoto continua valendo.

Uma das características da interface em Java é que as classes devem implementar todos os métodos definidos pela interface. Uma outra característica é que uma determinada classe pode implementar várias interfaces.

No exemplo abaixo, sobre TV's e Controle Remoto, é apresentado como interfaces são implementadas em Java.

9.2 Em Java.

O código abaixo implementa a interface ControleRemoto e as classes de Usuário, TV, TV29 e TV14.

```
public interface ControleRemoto{ //1
    public void Iliga_desliga(); //2
    public void Imuda_canal(String botao); //3
}
```

Código 9.1 - Interface ControleRemoto

Repare na linha 1, no código 8.1, a palavra reservada interface indicando que o que está sendo definido é uma interface.

```
public class Tv implements ControleRemoto{ //1
    //variáveis //2
    public int canal; //3
    public String liga_desliga; //4
    //Métodos //5
    public Tv() { //6
        this.canal = 3; //7
        this.liga_desliga = "desligado"; //8
    } //9
    public void lliga_desliga() { //10
        if (liga_desliga == "desligado")
            { liga_desliga = "liga"; } //11
        else { liga_desliga = "desligado"; } ; //12
    } //13
    public void Imuda_canal(String botao) { //14
        if (botao == "+") { canal++;} //15
        else { canal--;} //16
    } //17
} //18
```

Código 9.2 - Classe Tv.

No código 8.2 tem-se na linha 1 a palavra reservada implements, Isso significa que a classe TV irá implementar todos métodos definidos pela interface ControleRemoto, e isto é o que realmente ocorre, veja as linhas de 10 a 13 e de 15 a 17.

```
public class Tv29 extends Tv{
    public Tv29() {
        this.canal = 5;
        this.liga_desliga = "desligado";
    }
}
```

Código 9.3 - Classe Tv29.

O código 8.3 estende a classe `TV`. Assim a classe `TV29` é uma subclasse de `TV`. Desta forma os métodos especificados para a classe `TV` também se aplicam a classe `TV29`. Isso quer dizer que o comportamento implementado pela `TV` a partir da interface `ControleRemoto` se aplica também às classes de `TV29` e de `TV14`.

```
public class Tv14 extends Tv{
    public Tv14() {
        this.canal = 8;
        this.liga_desliga = "desligado";
    }
}
```

Código 9.4 - Classe Tv14.

O código 8.4 faz o mesmo com a classe `TV`.

```
public class ExecutaInterface {
    public static void main(String[] args) {
        Tv uma_tv = new Tv();
        uma_tv.liga_desliga();
        uma_tv.Imuda_canal("+");

        Tv uma_tv29 = new Tv29();
        uma_tv29.liga_desliga();
        uma_tv29.Imuda_canal("+");

        Tv uma_tv14 = new Tv14();
        uma_tv14.liga_desliga();
        uma_tv14.Imuda_canal("+");
    }
}
```

Código 9.5 - Classe ExecutaInterface

A classe usuário mostra que os métodos implementados pela classe `TV` a partir da interface `ControleRemoto` e que estes mesmo métodos, por herança, podem ficar disponíveis às demais subclasses.

9.3 Exercícios

1. Defina uma classe Radio e as classes RadioPortatil e RadiodeMesa de forma que operem com o mesmo ControleRemoto.

Capítulo 10 – Estudo de Caso.

10.1 O Caso.

Para exercitar os conceitos vistos nos capítulos anteriores considere o caso descrito a seguir.

Um coordenador das Faculdades Integradas Turing Machines resolveu automatizar o processo de orientação de projetos de seus professores com os respectivos alunos. Assim um professor(nome, sobrenome, matricula, titulação) oferece um ou mais projetos(código, descrição, data de início, data de término). Um projeto somente pode ter como responsável um único professor.

Os alunos(nome, sobrenome, matricula) se inscrevem em um ou mais projetos, sendo que os projetos podem ter um ou mais alunos.

A orientação é um processo que envolve o professor, o projeto e aluno. Quando da orientação o professor anota a data da orientação e as atividades previstas para as próximas etapas ou as atividades realizadas pelos alunos.

Um professor visitante (instituição de origem) também pode oferecer projetos nos mesmos moldes que um professor regular.

Analisando o caso acima se pode concluir que existem as seguintes classes:

- Professor com os seguintes variáveis: nome, sobrenome, matricula, titulação e métodos: construtor, mostraProfessor.
- ProfessorVisitante que herda de professor com as seguintes variáveis: faculdade_de_Origem e os métodos: construtor, mostraProfessor.
- Aluno com as seguintes variáveis: nome, sobrenome, matricula e com os métodos: construtor e mostraAluno.
- Projeto com as seguintes variáveis: código, descrição, dataInício, dataTérmino e com os seguintes métodos: construtor e mostraProjeto.
- Inscrição com as seguintes variáveis: um objeto de aluno e um objeto de projeto. Esta classe, em realidade, será uma composição das duas outras.
- Orientação com as seguintes variáveis: um objeto de professor, um objeto de aluno e um objeto de projeto, além das variáveis: dataOrientacao e DescricaoAtividades. Na realidade não se terá o objeto em si, mas apenas uma referência aos mesmos.

Entretanto outras variáveis devem estar presentes neste modelo: a de totalProjetosOferecidos, totalProfessoresCadastrados e totalProfessoresVisitantes, respectivamente para as classes Projetos, Professores e ProfessoresVisitantes. Estas variáveis não pertencem aos objetos em si mas sim as classes ao qual estes objetos representam. Assim devemos criar métodos de classe específicos para manipular com estas variáveis.

Assim temos os códigos 9.1 a 9.7 abaixo.

```
public class Aluno {
    // variáveis do objeto
    private String nome;
    private String sobreNome;
    private int matricula;
    // métodos do objeto
    public Aluno(String nome, String sobreNome,
        int matricula) {
        this.nome = nome;
        this.sobreNome = sobreNome;
        this.matricula = matricula;
    }
    public void mostraAluno() {
        System.out.println("nome: " + this.nome);
        System.out.println("nome familia: " + this.sobreNome);
        System.out.println("idade: " + this.matricula);
    }
}
```

Código 10.1 - Classe Aluno

```
public class Professor {
    // variáveis do objeto
    private String nome;
    private String nome;
    private String sobreNome;
    private String titulacao;
    private int matricula;
    // métodos do objeto
    public Professor(String nome, String sobreNome,String titulacao,
        int matricula) {
        this.nome = nome;
```

```

        this.sobreNome = sobreNome;
        this.titulacao = titulacao;
        this.matricula = matricula;
    }
    public void mostraProfessor() {
        System.out.println("nome: " + this.nome);
        System.out.println("nome familia: " + this.sobreNome);
        System.out.println("Titulação: " + this.titulacao);
        System.out.println("idade: " + this.matricula);
    }
    //métodos necessários devido a natureza PRIVATE das variáveis da
    classe
    public String onome() {
        return nome;
    }
    public String osobreNome() {
        return sobreNome;
    }
    public String atitulacao() {
        return titulacao;
    }
    public int amatricula() {
        return matricula;
    }
}

```

Código 10.2 - Classe Professor

```

public class ProfessorVisitante extends Professor {
    // variáveis do objeto
    private String nome;
    private String faculdadeOrigem;
    // métodos do objeto
    public ProfessorVisitante(String nome, String sobreNome,String
    titulacao, int matricula, String faculdadeOrigem) {
        super(nome, sobreNome, titulacao, matricula);
        this.faculdadeOrigem = faculdadeOrigem;
    }
    public void mostraProfessor() {

```

```

        System.out.println("nome: " + super.onome());
        System.out.println("nome familia: " + super.osobreNome());
        System.out.println("Titulação: " + super.atitulacao());
        System.out.println("idade: " + super.amatricula());
        System.out.println("Faculdade de origem: " +
this.faculdadeOrigem);
    }
}

```

Código 10.3 - Classe ProfessorVisitante

```

public class Projeto {
    // variáveis do objeto
    private String codigo;
    private String descricao;
    private String dataInicio;
    private String dataTermino;
    private Professor professorResponsavel;
    // métodos do objeto
    public Projeto(String codigo, String descricao,
        String dataInicio, String dataTermino, Professor
umProfessor) {
        this.codigo = codigo;
        this.descricao = descricao;
        this.dataInicio = dataInicio;
        this.dataTermino = dataTermino;
        this.professorResponsavel = umProfessor;
    }
    public void mostraProjeto() {
        System.out.println("nome: " + this.codigo);
        System.out.println("nome familia: " + this.descricao);
        System.out.println("idade: " + this.dataInicio);
        System.out.println("idade: " + this.dataTermino);
    }
}

```

Código 10.4 - Classe Projeto

```

public class Inscricao {

```

```

// variáveis do objeto
private Projeto umProjeto;
private Aluno umAluno;
// métodos do objeto
public Inscricao(Projeto umProjeto, Aluno umAluno) {
    this.umProjeto = umProjeto;
    this.umAluno = umAluno;
}
public void mostraInscricao() {
    System.out.println("nome: " + this.umProjeto);
    System.out.println("nome familia: " + this.umAluno);
}
}

```

Código 10.5 - Classe Inscricao

```

public class Orientacao {
    // variáveis do objeto
    private Inscricao umaInscricao;
    private String dataOrientacao;
    private String descricaoOrientacao;
    // métodos do objeto
    public Orientacao(Inscricao umaInscricao, String dataOrientacao,
String descricaoOrientacao) {
        this.umaInscricao = umaInscricao;
        this.dataOrientacao = dataOrientacao;
        this.descricaoOrientacao = descricaoOrientacao;
    }
    public void mostraOrientacao() {
        System.out.println("Inscricao: " + this.umaInscricao);
        System.out.println("Data da orientação: " +
this.dataOrientacao);
        System.out.println("Descrição orientação: " +
this.descricaoOrientacao);
    }
}

```

Código 10.6 - Classe Orientacao

```

public class ExecutaCaso {
    public static void main (String args[]) {
        // criando os professores
        Professor joseProfessor = new Professor ("Jose", "Santos",
"MSc", 2345);
        joseProfessor.mostraProfessor();
        Professor joaoProfessor = new Professor ("Maria", "Clara",
"DSc", 4567);
        joaoProfessor.mostraProfessor();

        ProfessorVisitante    willianProfessor    =    new
ProfessorVisitante ("Willia", "Blake", "DSc", 2346, "U of Sidney");
        willianProfessor.mostraProfessor();

        //criando os projetos - repare que é passada a referência
do professor responsável pelo projeto
        Projeto proj1 = new Projeto("abc123", "Gerencia de
requisitos", "12set2005", "12dez2005", joseProfessor);
        proj1.mostraProjeto();

        Projeto proj2 = new Projeto("def345", "Gerencia de
configuração", "12ago2005", "12nov2005", joaoProfessor);
        proj2.mostraProjeto();

        Projeto proj3 = new Projeto("ghil23", "Gerencia de riscos",
"10jul2005", "12out2005", willianProfessor);
        proj3.mostraProjeto();

        // criando os alunos
        Aluno carlosAluno = new Aluno ("Carlos", "Miguel", 5678);
        carlosAluno.mostraAluno();

        Aluno pauloAluno = new Aluno ("Paulo", "Antunes", 3245);
        pauloAluno.mostraAluno();

        Aluno joselioAluno = new Aluno ("Joselio", "Silva", 9876);
        joselioAluno.mostraAluno();

        // fazendo a inscrição dos alunos nos projetos

```

```

Inscricao inscricao01 = new Inscricao (proj1, carlosAluno);
inscricao01.mostraInscricao();

Inscricao inscricao02 = new Inscricao (proj2, pauloAluno);
inscricao02.mostraInscricao();

//fazendo a orientação
Orientacao orientacao01 = new Orientacao (inscricao01,
"15set2005", "objetivos do projeto");
orientacao01.mostraOrientacao();

Orientacao orientacao02 = new Orientacao (inscricao01,
"21set2005", "entrega do escopo");
orientacao02.mostraOrientacao();

Orientacao orientacao03 = new Orientacao (inscricao01,
"03out2005", "definição dos casos de uso");
orientacao03.mostraOrientacao();
    }
}

```

Código 10.7 - Classe ExecutaCaso

Parte Dois – Introdução aos princípios de projeto OO: padrões.

Capítulo 11 – Princípios e Padrões.

De forma resumida o desenho e a programação de sistemas orientados a objetos pode ser definido como a identificação de classes, a identificação de métodos e a especificação de mensagens entre estas classes. Esta forma de abordar o problema encobre várias dificuldades encontradas por aqueles que iniciam na programação orientada a objetos.

Foi visto, na parte um deste trabalho, que aprender a utilização da linguagem JAVA é apenas um começo. Saber como identificar, especificar e projetar orientado a objetos é um outro problema. A identificação e a especificação não são assunto no escopo deste trabalho. Assim, será tratada apenas a questão de projeto, em particular da questão de padrões de projeto e dentre deles àqueles que estão diretamente associados a o projeto básico.

Desta forma, é necessária a caracterização de princípios e questões de desenho que devem ser considerados. Decidir quais os métodos devem ser alocados a classes é uma tarefa que deve ser feita de forma criteriosa, pois impacta seriamente no desenho final do sistema.

Para a alocação de métodos às classes o conceito de distribuição de responsabilidade [Larman, 2004] é importante. A distribuição de responsabilidades considera as obrigações e o comportamento que um determinado objeto deve possuir. As responsabilidades são caracterizadas de duas formas: conhecer e fazer.

As *responsabilidades de fazer* consideram tanto o controle de outros objetos quanto saber o que o próprio objeto deve saber. As *responsabilidades de conhecer* consideram saber os dados armazenados e a relação com outros objetos. Uma boa modelagem de classes ilustra bem as *responsabilidades de conhecer* que devem ser atribuídas entre as classes de objetos.

Deve-se destacar que uma determinada responsabilidade pode ser representada em vários métodos espalhados pelas classes de objetos. A quantidade de métodos que implementam uma responsabilidade define a granularidade da mesma. Este aspecto diferencia responsabilidade de método, as responsabilidades são abstratas. Outro aspecto que merece ênfase é que a responsabilidade considera a questão da colaboração entre os objetos através de seus métodos.

Para o emprego de responsabilidades Larman[2004] definiu alguns princípio ou padrões que auxiliam nesta tarefa: os padrões GRASP – General Responsibility Assignment Software Patterns (Padrões Gerais de Software para Alocação de Responsabilidade). Nesta parte serão considerados cinco destes princípios: especialista da informação, o especialista de criação, baixo acoplamento, alta coesão e controlador.

Mas o que é um padrão? “Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos

reutilizável” [Gamma et all, 1995, pág 20]. Um padrão não desceve algo novo, sua importância está na sistematização de conhecimento de projeto para a solução de determinados problemas em um determinado contexto. Desta forma a descrição possui quatro elementos básicos: o nome do padrão, o problema a ser resolvido, a solução proposta e as consequências das vantagens e desvantagens de se aplicar o padrão. Um dos objetivos dos padrões é o de aprender com o sucesso dos outros ao invés dos fracassos [Eckel, 2003 apud Mark Johnson].

Costuma-se falar de padrões associados à implementação, entretanto, o modelo de apresentar soluções padronizadas para determinados problemas atualmente está presente em várias esferas da construção de software tais como: casos de uso, análise, UML entre outros.

11.1 O padrão “Especialista da Informação”.

O problema que o padrão Especialista da Informação procura resolver é: quem conhece os dados e informações de um determinado objeto? Este padrão é um dos princípios básicos de todos os padrões, pois lida com o princípio básico de distribuição de responsabilidade pelas classes. A solução para este problema está em atribuir a responsabilidade a classe que detenha ou obtenha a informação necessária a sua utilização através de seu acesso direto.

Exemplos:

- Em sistema de pedidos qual classe deve ser responsável pelo cálculo do valor final do pedido? Considerando as classes Pedido e Item de Pedido. Podemos considerar que Pedido é a classe que deve possuir o método `valorFinaldoPedido()` já que é ela que possui acesso imediato a todos os itens de pedido.
- Em um sistema acadêmico qual classe deveria ser responsável pelo cálculo de aprovação de uma determinada turma? Considerando as classes Turma, Notas de Alunos. Podemos considerar que Turma é a classe que deve possuir o método `calculaAprovacaoTurma()` já que ela possui acesso imediato às notas dos alunos na Turma.

Como identificar que uma classe possui a responsabilidade de acesso para a obtenção de informação de outra classe quando se possuem várias classes? A solução é verificar o modelo conceitual de dados. É este modelo que irá representar as principais relações entre as classes e objetos. Estas relações podem ser apenas estruturais como uma relação de associação, como também pode ser uma relação de posse completa, como a relação de composição, ou de posse parcial como a relação de agregação.

11.2 O padrão de “Criação”.

O problema que o padrão de Criação procura resolver é: quem deve criar um determinado objeto? A princípio, várias abordagens podem ser consideradas já que existem várias classes de objetos que

compõem o seu modelo do problema. Entretanto, deve-se atribuir a uma destas classes a responsabilidade de criação de um determinado objeto, logo reina a pergunta listada acima.

Para aplicar o padrão de Criação deve-se:

- Manter o princípio do padrão Especialista, mantendo as responsabilidades associadas às classes de objetos de acesso mais imediato a este.
- Atribuir a responsabilidade de criação à classe de objetos que possua uma associação com a outra classe de objetos, que necessita ser criada. Existem vários tipos de associação, tais como:
 - De uma para muitos, exemplo: uma **venda** possui vários **itens de venda**, sendo **venda** e **itens de venda** classes;
 - De composição, exemplo: um **pedido** é composto de uma ou mais **formas de pagamento**, sendo **pedido** e **formas de pagamento** classes;
 - De agregação, exemplo: uma **bicicleta** possui dois **pneus**, onde **bicicleta** e **pneus** são classes.
- A classe de objetos que cria outra classe de objetos já deve existir. Desta forma associações de *um para muitos* são uma evidência de qual classe deve criar a outra.

Wazlawick[2004] estabelece um conjunto de verificações para se decidir qual classe deverá criar instâncias em outra classe, são estes:

- Verificar se a classe de objeto é parte da uma relação de agregação ou composição. Se for o caso o criador será a classe de objeto agregadora.
- Se a verificação i falhar certifique-se que a classe possui uma associação de um para muitos ou de um para um-zero para a classe do objeto a ser criado. Se existir e estiver em um caminho possível de uma classe controladora, esta poderá ser a classe criadora.
- Em caso de empate verifique qual a classe está mais fortemente associada a classe a ser criada.

11.3 O padrão “Baixo Acoplamento”.

O padrão baixo acoplamento tem como objetivo a redução e minimização da conexão de um objeto com outros. Esta conexão esta relacionada a ter conhecimento sobre os outros objetos ou simplesmente depender de outros objetos. De forma sujacente manter o acoplamento baixo significa redizir o impacto de mudança já que ao se alterar um objeto um mínimo de outros objetos serão afetados.

Desta forma a atribuição de responsabilidade de ser tal que mantenha as visibilidades entre os objetos em seu nível mínimo.

11.4 O padrão “Alta Coesão”.

O padrão de alta coesão está diretamente associado ao padrão de baixo acoplamento. A problema que se procura resolver é de como manter o objeto devidamente focado naquilo que ele se propõem a realizar. Assim as responsabilidades de um determinado objeto devem ser aquelas reduzam a complexidade do mesmo. Uma classe, com baixa coesão, provavelmente estará representando mais de um conceito [Wazlawick, 2004].

11.4 O padrão “Controlador”.

O padrão controlador procura tratar do problema de qual objeto deve coordenar as atividades de um adeterminada operação do sistema. Em geral este objeto pode ser o próprio objeto sistema ou o objeto que representa a funcionalidade em que está sendo implementada.

11.5 Exemplo um.

O código 11.1, de forma simplificada, servirá de ponto de partida para ilustrar os diversos padrões que podem ser utilizados. Um exemplo com interface gráfica e utilizandoo padrão MVC pode ser encontrado no endereço: <http://csis.pace.edu/~bergin/mvc/mvcgui.html>.

```
public class ConversaoTemp
{
    private double fahrenheit = 32.0;

    public double getFahrenheit(){return fahrenheit;}
    public double getCelsius(){return (fahrenheit - 32.0)*5.0/9.0;}
    public void setFahrenheit(double tempF)
    {
        fahrenheit = tempF;
    }
    public void setCelsius(double tempC)
    {
        fahrenheit = tempC*9.0/5.0 + 32.0;
    }
    public static void mostraFahrenheit (double tempF)
    {System.out.println ("O valor de Farenheit é " + tempF);
    }
    public static void mostraCelsius (double tempC)
    {System.out.println ("O valor de Farenheit é " + tempC);
    }
    public static void mostraEscalaCelsius (ConversaoTemp temp)
    {for (int i=0; i<101; i++)
        { temp.setCelsius(i);
          System.out.println("Valor de Celsius" + i + "=" +
            temp.getFahrenheit() );}
    }
}
```

Código 11.1 - Classe ConversaoTemp

O exemplo do código 11.1 tem como objetivo realizar a conversão de Celsius para Farenheit e vice-versa. No exemplo acima nota-se que existem duas responsabilidades básicas associadas a classe

ConversãoTemp: a primeira que corresponde ao modelo da própria conversão de Celsius para Fahrenheit e outra de mostrar seus valores. Assim, a classe não está coesa, ou seja, está com baixa coesão sendonecessário separar essas duas responsabilidades básicas. As classes divididas seriam então representadas nos códigos 11.2 e 11.3. O código 11.2 fica com a responsabilidade de apenas representar o modelo e o código 11.3 com a responsabilidade de mostrar os valores

```
public class NovaConversaoTemp
{
    private double fahrenheit = 32.0;

    public double getFahrenheit(){return fahrenheit;}
    public double getCelsius(){return (fahrenheit - 32.0)*5.0/9.0;}
    public void setFahrenheit(double tempF)
    {
        fahrenheit = tempF;
    }
    public void setCelsius(double tempC)
    {
        fahrenheit = tempC*9.0/5.0 + 32.0;
    }
}
```

Código 11.2 - Classe NovaConversaoTemp

```
public class MostraConversaoTemp
{
    public static void mostraFahrenheit (double tempF)
    {System.out.println ("O valor de Fahrenheit é " + tempF);}
    }
    public static void mostraCelsius (double tempC)
    {System.out.println ("O valor de Fahrenheit é " + tempC);}
    }
    public static void mostraEscalaCelsius (NovaConversaoTemp temp)
    {for (int i=0; i<101; i++)
        { temp.setCelsius(i);
          System.out.println("Valor de Celsius" + i + "=" +
            temp.getFahrenheit() );}
    }
}
```

Código 11.2 - Classe MostraConversaoTemp

Agora a classe DemoConversaoTemp irá criar o objeto da Classe NovaConversaoTemp e mostrar seus valores, conforme o código 11.3.

```
public class DemoConversaoTemp
{
    public static void main (String args[]) {
        NovaConversaoTemp objetoTemp = new NovaConversaoTemp();
        double F = objetoTemp.getFahrenheit();
        MostraConversaoTemp.mostraCelsius (F);
        MostraConversaoTemp.mostraEscalaCelsius (objetoTemp);
    }
}
```

Código 11.3 - Classe DemoConversaoTemp

Entretanto verifica-se que o acoplamento entre as classes `MostraConversaoTemp` e `NovaConversaoTemp` se estabelece através de dois modos de passagem de parâmetro. Pode-se reduzir este acoplamento passando apenas o objeto que se deseja mostrar e deixando para a classe `MostraConversaoTemp` os encargos de tratar o objeto da forma que lhe convier. Uma outra alternativa é utilizar métodos polimórficos.

Assim o novo código desta última classe é mostrado em 11.4. A nova chamada das classes é mostrado em 11.5.

```
public class NovoMostraConversaoTemp
{
    public static void mostraFahrenheit (NovaConversaoTemp temp)
    {System.out.println ("O valor de Fahrenheit é " +
temp.getFahrenheit());
    }
    public static void mostraCelsius (NovaConversaoTemp temp)
    {System.out.println ("O valor de Celsius é " +
temp.getCelsius());
    }
    public static void mostraEscalaCelsius (NovaConversaoTemp temp)
    {for (int i=0; i<101; i++)
    { temp.setCelsius(i);
    System.out.println("Valor de Celsius" + i + "=" +
temp.getFahrenheit() );}
    }
}
```

Código 11.4 - Classe NovoDemoConversaoTemp

```
public class NovoDemoConversaoTemp
{
    public static void main (String args[]) {
        NovaConversaoTemp objetoTemp = new NovaConversaoTemp();
        NovoMostraConversaoTemp.mostraCelsius (objetoTemp);
        NovoMostraConversaoTemp.mostraEscalaCelsius
(objetoTemp);
    }
}
```

Código 11.5 - Classe NovoDemoConversaoTemp

Referências bibliográficas.

Bruce Eckel Thinking in Java. 4rd edition.

Este é um excelente livro. Possui uma leitura fácil com exemplos ilustrativos e claros. Pode ser baixado da internet gratuitamente ou comprado como um livro regular. Está em inglês. Disponível em <http://mindview.net/Books>.

Bruce Eckel .Thinking in Patterns. Revision 0.9 - May 20, 2003

Disponível em <http://mindview.net/Books/TIPatterns/>

Craig Larman. Applying UML and Patterns—An Intro to OOA/D and Iterative Development. 3rd edition. 2004.

É um ótimo livro com vários fundamentos de UML e padrões. O capítulo 1 de orientação a objetos está disponível em: http://www.craiglarman.com/book_applying/intro.pdf.

Erick Gamma, Richard Helm, Ralph Johnson e John Vlissides. Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos.

Este é um livro clássico de padrões de projeto. Difícil de ler para quem está iniciando o assunto. Para iniciantes o melhor é o do Larman.

H. M. Deitel e P.J. Deitel. Java How to Program, 6/e Prentice Hall 2005 ou Java como programar. Bookman.

Este é um clássico do Java. Acredito que este livro não deva ser o primeiro livro que um novato devesse comprar, mas é uma referência completa. Algumas downloads do **Java How to Program, 6/e** estão disponíveis em <http://www.deitel.com/books/jHTP6/>. No site pode-se baixar os seguintes capítulos do livro: Chapter 1: Introduction to Computers, the Internet and World Wide Web; Chapter 2: Introduction to Java Programming; Chapter 3: Introduction to Classes and Objects

Java Tutorial da Sun.

Este tutorial da Sun é muito bom. Cobre grande parte da tecnologia Java. Além de poder realiza-lo pela internet este pode ser baixado e utilizado localmente. Disponível em <http://java.sun.com/docs/books/tutorial/>

Rafael Santos. Introdução a Orientação a Objetos usando Java.

Este é um dos livros da série SBC/Campus. É um ótimo livro, bem explicado e com muitos exemplos e que procura explicar os conceitos de OO em Java. É um excelente livro introdutório ao assunto.

Raul Sidnei Wazlawick. Análise e Projeto de Sistemas de Informação Orientados a Objetos. 2004.

Este é um outro livro da série SBC/Campus. É um ótimo livro de análise e projeto de sistemas de informação. Muito prático e objetivo.