

Sistema Aberto de Educação



Guia de Estudo

Programação Orientada a Objetos I

2ª Parte



Instituição Credenciada pelo MEC
Centro Universitário do Sul de Minas



SABE – Sistema Aberto de Educação

**Av. Cel. José Alves, 256 - Vila Pinto
Varginha - MG - 37010-540
Tele: (35) 3219-5204 - Fax - (35) 3219-5223**

Instituição Credenciada pelo MEC – Portaria 4.385/05

**Centro Universitário do Sul de Minas - UNIS/MG
Unidade de Gestão da Educação a Distância – GEaD**

**Mantida pela
Fundação de Ensino e Pesquisa do Sul de Minas - FEPESMIG**

Varginha/MG

Todos os direitos desta edição reservados ao Sistema Aberto de Educação – SABE.
É proibida a duplicação ou reprodução deste volume, ou parte do mesmo, sob
qualquer meio, sem autorização expressa do SABE.

005.13
M188P Magalhães, Demétrio Reno.

Guia de Estudo – Programação Orientada
a Objetos I. Demétrio Renó Magalhães.
Varginha: GEaD-UNIS/MG, 2009.

95p.

1. Objetos. 2. Classe. 3. Polimorfismo. 4.
Herança. 5. Métodos. 6. Atributos I. Título.

REITOR

Prof. Ms. Stefano Barra Gazzola

GESTOR

Prof. Ms. Wanderson Gomes de Souza

Supervisora Técnica

Profª. Ms. Simone de Paula Teodoro Moreira

Coord. do Núcleo de Recursos Tecnológicos

Lúcio Henrique de Oliveira

Coord. do Núcleo de Desenvolvimento Pedagógico

Profª. Vera Lúcia Oliveira Pereira

Revisão ortográfica / gramatical

Profª. Maria José Dias Lopes Grandchamp

Design/diagramação e Equipe de Tecnologia Educacional

Prof. Celso Augusto dos Santos Gomes

Profª. Débora Cristina Francisco Barbosa

Danúbia Pinheiro Teixeira












Jacqueline Aparecida Silva

Autor(a)

Demétrio Renó Magalhães

Graduado em Matemática Aplicada a Informática pela Fundação de Ensino e Pesquisa de Itajubá (FEPI) e Mestre em Engenharia Elétrica pela Escola Federal de Engenharia de Itajubá (EFEI). Professor no UnilesteMG nos cursos de Computação - Sistemas de Informação e Engenharias Mecânica, Elétrica, Produção, Sanitária e Ambiental e Materiais, pesquisador do Laboratório de Sistemas de Tempo Real (LTR), coordenador de pesquisa no curso de Computação – Sistemas de Informação, coordenador do Laboratório de Inteligência Computacional e Instrutor de Treinamentos.

TABELA DE ÍCONES

	REALIZE. Determina a existência de atividade a ser realizada. Este ícone indica que há um exercício, uma tarefa ou uma prática para ser realizada. Fique atento a ele.
	PESQUISE. Indica a exigência de pesquisa a ser realizada na busca por mais informação.
	PENSE. Indica que você deve refletir sobre o assunto abordado para responder a um questionamento.
	CONCLUSÃO. Todas as conclusões, sejam de idéias, partes ou unidades do curso virão precedidas desse ícone.
	IMPORTANTE. Aponta uma observação significativa. Pode ser encarado como um sinal de alerta que o orienta para prestar atenção à informação indicada.
	HIPERLINK. Indica um link (ligação), seja ele para outra página do módulo impresso ou endereço de Internet.
	EXEMPLO. Esse ícone será usado sempre que houver necessidade de exemplificar um caso, uma situação ou conceito que está sendo descrito ou estudado.
	SUGESTÃO DE LEITURA. Indica textos de referência utilizados no curso e também faz sugestões para leitura complementar.
	APLICAÇÃO PROFISSIONAL. Indica uma aplicação prática de uso profissional ligada ao que está sendo estudado.
	CHECKLIST ou PROCEDIMENTO. Indica um conjunto de ações para fins de verificação de uma rotina ou um procedimento (passo a passo) para a realização de uma tarefa.
	SAIBA MAIS. Apresenta informações adicionais sobre o tema abordado de forma a possibilitar a obtenção de novas informações ao que já foi referenciado.
	REVENDO. Indica a necessidade de rever conceitos estudados anteriormente.

SUMÁRIO

UNIDADE III.....	6
3 Herança	6
3.1 Representação na UML	8
3.2 Um construtor que chama outro construtor.....	9
3.3 Exercícios	10
3.4 Sobrecarga de métodos.....	11
3.6 Chamando o método que foi reescrito.	13
3.7. Herança Múltipla	13
3.8 Classes Abstratas	15
3.8.1 O poder da abstração	16
3.9 Exercícios	19
UNIDADE IV	20
4 Encapsulamento	20
4.1 Modificadores de acesso.....	20
4.1.1 Modificador public	20
4.1.2 Modificador private.....	21
4.1.3 Modificador de acesso - Protected	23
4.1.4 static.....	25
4.2 Exercícios	27
UNIDADE V	34
5 Interface.....	34
5.1 Interfaces são usadas para:	35
5.2 Palavra chave final	35
5.2.1 Uma classe final.....	35
5.2.2 Exercícios.....	37
5.3 Polimorfismo.....	38
5.3.1 Por que usar polimorfismo?	40
5.4 Exercícios	42
6. Respostas dos exercícios	43
REFERÊNCIAS	93

UNIDADE III

Ao final da aula o aluno será capaz de:

- Entender o conceito de herança;
- Entender o relacionamento é um;
- Escrever classes que usem herança;
- Utilizar o modificador de acesso protected.
- Entender a sobrecarga de métodos;
- Entender a vantagem da reescrita de métodos;
- Diferenciar sobrecarga de reescrita de métodos;
- Reescrever métodos em subclasses;
- Utilizar métodos reescritos;
- Entender a abstração de dados;
- Declarar classes abstratas;
- Declarar métodos abstratos;
- Herdar classes abstratas;
- Reescrever métodos abstratos;
- Utilizar o poder da abstração.

3 Herança


Na orientação a objetos podemos relacionar classes de forma que uma delas herda "tudo" que a outra tem sem copiar e colar o código. Isso é chamado de relação de classe mãe e classe filha.

Em Java a palavra `extends` representa essa herança. Vejamos:

	<pre>class Empregado{ String nome; double salario; }</pre>
---	--

Essa classe representa qualquer empregado de uma empresa e na empresa temos vários tipos de empregados, cada um com suas atividades específicas. Um gerente **é um** funcionário que necessita de atributos específicos para que exerça sua função como uma senha por exemplo.

A classe Gerente pode ser modelada da seguinte maneira. Vejamos:

	<pre>class Gerente{ String nome; double salario; int senha; public boolean autentica(int senha){ if(this.senha == senha){ System.out.println("Acesso permitido"); } } }</pre>
---	---


	<pre> return true; } else{ System.out.println("Acesso negado"); return false; } } }</pre>
--	--

A classe Gerente repetiu os atributos da classe Empregado e isso não é uma boa prática de programação. Poderíamos então fazer o seguinte: deixaríamos a classe Empregado mais genérica, mantendo nela a senha de acesso, caso o empregado não fosse um gerente, deixaríamos esse atributo vazio. E os métodos de autenticação a classe Gerente não seriam utilizados na classe Empregado se ele não fosse um Gerente. Isso também não é uma boa pois a classe fica confusa com campos vazios e sem função nenhuma dependendo da situação.

Para resolver este problema utilizamos a técnica da **herança**. Vamos fazer com que Gerente seja uma extensão de Empregado. Neste caso teremos as classes Empregado e Gerente da seguinte forma. Vejamos:

	<pre>class Empregado{ String nome; double salario; }</pre>
---	--

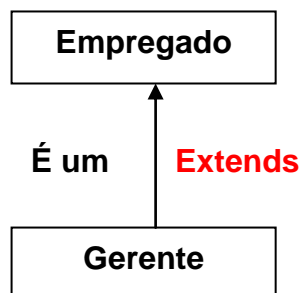
E a classe Gerente:

	<pre>class Gerente extends Empregado{ int senha; public boolean autentica(int senha){ if(this.senha == senha){ return true; } else{ return false; } } }</pre>
---	--

No exemplo acima fizemos com que a classe Gerente se tornasse uma extensão da classe Empregado utilizando a palavra **extends**.

Sendo assim, agora toda vez que criarmos um objeto do tipo Gerente (ou instanciarmos a classe Gerente), este objeto também possuirá os atributos e métodos da classe Empregado, pois agora Gerente **é um** Empregado.

3.1 Representação na UML




Para instanciarmos um Gerente fazemos:

	<pre> class TestaGerente{ public static void main (String[] args){ //instanciando um objeto Gerente referenciado por g Gerente g = new Gerente(); //utilizando os atributos e métodos do objeto Gerente g.nome = "Gustavo"; g.salario = 5434.34f; if(g.autentica(234) == true){ System.out.println("Acesso autorizado!"); } else { System.out.println("Acesso negado!"); } } } </pre>
--	---


No exemplo acima, a classe Gerente também conhecida como classe **filha** ou **subclasse** herda todos os atributos e métodos da classe Empregado também conhecida como **superclasse** ou **classe mãe**.

Vamos inserir um construtor na classe Empregado. O código ficará assim:

	<pre>class Empregado{ String nome; double salario; Empregado(String nome, double salario){ this.nome = nome; this.salario = salario; } }</pre>
---	--

O construtor da classe acima atribui aos atributos da classe um nome e um salário que são usados como parâmetros no construtor da classe Empregado.


3.2 Um construtor que chama outro construtor

	<pre>class Gerente extends Empregado{ int senha; Gerente (String n, double s){ super (n, s) ; } public boolean autentica(int senha){ if(this.senha == senha){ return true; } else{ return false; } } }</pre>
---	--

O código acima mostra que a classe Gerente deve ser instanciada utilizando no construtor o nome e o salário (que são argumentos do construtor da classe Empregado). Observe que a palavra **super** tem como argumentos **n** e **s**. Esses argumentos serão "passados" como argumentos para o construtor da super classe, nesse caso Empregado.

Quando utilizamos a palavra **super** e abrimos um parênteses, onde podemos ou não utilizar argumentos, e depois fechamos os parênteses estamos nos referindo ao construtor da super classe dessa classe. Se usarmos a palavra **super**, um separador **ponto** e um nome de um **método** estaremos nos referindo a um método da super classe.

Agora quando quisermos criar um Gerente, obrigatoriamente teremos que dar um nome e um salário para esse Gerente. Vejamos o código abaixo:

	<pre>class TestaGerente{ public static void main (String[] args){ //instanciando um objeto Gerente referenciado por g Gerente g = new Gerente("Gustavo", 5645,44f); if (g.autentica(234) == true){ System.out.println("Acesso autorizado!"); } else { System.out.println("Acesso negado!"); } } }</pre>
---	---

3.3 Exercícios

1. Escreva uma classe chamada ClasseA que possua um método chamado metodoA que retorna "eu sou o método A".
2. Escreva uma classe chamada ClasseB que estende da ClasseA e possua um método chamado metodoB que retorna "eu sou o método B".
3. Crie uma aplicação, em Java, que instancie um objeto da ClasseB e imprima utilizando o console o retorno dos dois métodos criados.
4. Crie uma classe chamada Animal que possua os atributos peso e cor. O construtor dessa classe deverá atribuir o peso e a cor para o animal no momento da instanciação. Crie os métodos para configurar e ler esses atributos (métodos set e get). Um método chamado mostraAnimal deverá ser criado para mostrar o peso do animal e a cor do animal. O método mostraAnimal deverá escrever na janela gráfica "O animal tem cor " +cor+ "e pesa: " +peso+ " kilos."
5. Crie uma classe chamada Cachorro que estende a classe Animal e coloque os atributos nome e raça. Crie um construtor para atribuir o nome, raça, peso e cor do animal. No corpo do construtor chame o construtor da superclasse para atribuir peso e cor ao animal instanciado. Crie um outro construtor que atribua somente o nome do animal ao atributos nome no momento em que o objeto for instanciado. Crie métodos para configurar e mostrar os atributos nome e raça.
6. Crie uma classe chamada Cox que estende de Cachorro. Coloque como atributo tosa do tipo booleano. Crie um construtor que tenha como argumentos o peso, cor e tosa. Os argumentos peso e cor deverão ser passados para o construtor da super classe e tosa deverá ser atribuído ao atributo tosa. Crie um método chamado mostraAnimal que deverá escrever na janela "Este cachorro está tosado!" se tosa for verdadeiro (true) e a frase "Você precisa tosar o cachorro!" se tosa for falsa (false).

7. Implemente o exercício anterior no Java criando uma janela gráfica com os campos peso, cor e tosa. Tosa pode ser um *combobox*. Um botão de mostrar os dados deverá ser implementado para que em uma outra janela mostre os dados preenchidos pelo usuário.

3.4 Sobrecarga de métodos

Os métodos, tanto operacionais quanto os construtores podem ser sobrecarregados. Sobrecarga de método significa que podemos ter mais de um método dentro da mesma classe com **mesmo nome**, mas com **assinatura** diferente.

O que é **assinatura de um método**? A assinatura de um método são seus parâmetros. A assinatura se difere uma da outra quanto ao número de parâmetros e posição dos parâmetros. Vejamos:



```
setRelatorio(int nota, String nome);  
setRelatorio(String nome, int nota);  
setRelatorio(String nome, char serie, int nota);  
setRelatorio(char serie, String nome);
```

Os códigos acima são exemplos de sobrecarga de método, nesse caso o método `setRelatorio` que possui 4 versões. Observe que a ordem dos parâmetros não são as mesmas e a quantidade também muda. Isso é chamado de **sobrecarga de métodos**

Os métodos construtores também podem ser sobrecarregados.

3.5 Reescrita de métodos

Voltaremos ao nosso exemplo da classe `Empregado`. Supondo que os empregados recebam uma bonificação da seguinte forma: Os empregados (comuns) receberão 10% do valor do seu salário de bônus e os gerentes receberão 15% do valor do seu salário como bônus.


Vejamos como fica o método **`getBonificacao()`**




```
class Empregado{  
    String nome;  
    double salario;  
    public double getBonificacao(){  
        return this.salario * 0.10;  
    }  
}
```

	<pre> } } </pre>
--	------------------------

Vejamos a classe Gerente:

	<pre> class Gerente extends Empregado{ int senha; public boolean autentica(int senha){ if(this.senha == senha){ return true; } else{ return false; } } } </pre>
---	---


Se deixarmos a classe Gerente como está ela herdar o método `getBonificacao()` com o valor de 10% e não de 15%. Neste caso a classe Gerente tem que **reescrever** o método `getBonificacao()` da classe `Empregado`. Isso é feito da seguinte forma:

	<pre> class Gerente extends Empregado{ int senha; public boolean autentica(int senha){ if(this.senha == senha){ return true; } else{ return false; } //reescrevendo o método getBonificacao() public double getBonificacao(){ return this.salario * 0.15; } } } </pre>
---	---

Agora ao instanciarmos um objeto de Gerente e acessarmos o método `getBonificacao()` ele atribuirá 15% ao salário. Pois o método que será acionado é o método da classe Gerente e não da classe Empregado.

3.6 Chamando o método que foi reescrito.

Podemos chamar um método que foi reescrito (método original) utilizando a palavra **super** seguida pelo separador ponto (.) e o nome do método. Vejamos:

	<pre>class Gerente extends Empregado{ int senha; public double getBonificacao(){ return super.getBonificacao() + 1000; } }</pre>
---	--

A classe Gerente acima reescreveu o método `getBonificacao()` adicionando 1000 a bonificação recebida por um funcionário. A palavra **super** indica que o método `getBonificacao()` utilizado foi o da classe Empregado que é a **superclasse** de Gerente.

3.7. Herança Múltipla

A herança múltipla é pode ser implementada em algumas linguagens de programação orientada a objetos (em Java não temos herança múltipla). Temos uma herança múltipla quando temos uma classe filha que herda de duas ou mais classes mãe simultaneamente.

O uso intensivo de herança múltipla torna o código confuso. Phython e Linguagem C++ implementam herança múltipla.

Java não implementa a herança múltipla mas pode implementar interfaces.

Exercícios

1. Qual a vantagem da reescrita de métodos?
2. O método de qual classe será chamado pelo comando `super.NomeDoMetodo()`?

Exercícios complementares

1. Escreva uma classe chamada Pedido. A classe Pedido deverá possuir os atributos privados: `cliente`, `numeroDoPedido`, `quantidadeDeItens`. Crie um construtor que utilize como parâmetro o nome do cliente. Um outro construtor deverá possuir o nome do cliente e o número do pedido e um outro construtor

deverá possuir como argumento o nome do cliente a quantidadeDeltens e o número do pedido. Insira os métodos para mostrar cada atributo da classe Pedido.

2. Escreva uma aplicação que instancie 3 objetos sendo que cada um utilize um construtor criado. Mostre todos os atributos para cada objeto instanciado.
3. Crie uma classe Conta, que possua um saldo, e os métodos para configurar saldo, depositar, e retirar. Siga o esqueleto abaixo:

```
class Conta {  
    private double saldo;  
    void deposita(double x) {  
        //..  
    }  
    void retira(double x) {  
        //..  
    }  
    double getSaldo() {  
        //..  
    }  
}
```

4. Preencha o esqueleto conforme exemplos vistos no decorrer dos módulos. Crie o método retira() que retira um valor da conta e retorne um boolean indicando o sucesso da operação.
5. Adicione um método na classe Conta, que atualiza essa conta de acordo com a taxa selic fornecida.

```
void atualiza(double taxaSelic) {  
    this.saldo = this.saldo * (1 + taxaSelic);  
}
```

6. Crie duas subclasses da classe Conta: ContaCorrente e ContaPoupanca. Ambas terão o método atualiza reescrito: A ContaCorrente deve atualizar o saldo com o dobro da taxaSelic, porém subtraindo 15 reais de taxa bancária. A ContaPoupanca deve atualizar-se com 75% da taxaSelic. Além disso, a ContaCorrente deve reescrever o método deposita, afim de retirar a CPMF de 0.038% a cada depósito.

```
class ContaCorrente extends Conta {  
    void atualiza(double taxaSelic) {  
        // fazer conforme enunciado  
    }  
    void deposita(double valor) {  
        // o deposito deve depositar apenas 99.62% do valor  
    }  
}
```

```
}  
  
class ContaPoupanca extends Conta {  
    void atualiza(double taxaSelic){  
        // fazer conforme enunciado  
    }  
}
```


7. Crie uma aplicação que instancie essas classes, atualize-as e veja o resultado.

3.8 Classes Abstratas

A abstração é uma maneira utilizada pelo programador de uma classe mãe (superclasse) forçar o programador da classe filha (subclasse) a definir um comportamento (o que seus métodos irão fazer).

Um método com funcionalidade (tem um código no corpo do método) definido em uma superclasse é o chamado de comportamento **padrão** porque especifica o mesmo comportamento que as subclasses irão seguir. Quando fazemos uma herança, isto é, criamos uma subclasse podemos optar por utilizar o comportamento padrão da superclasse ou redefiní-lo de acordo com suas necessidades na subclasse (**reescrita de métodos**).

Vejamos a classe abaixo:

	<pre>class Gerente extends Empregado{ int senha; public boolean autentica(int senha){ if(this.senha == senha){ return true; } else{ return false; } //reescrivendo o método getBonificacao() public double getBonificacao(){ return this.salario * 0.15; } }</pre>
---	--

Você pode observar que o método `getBonificacao()` da classe `Empregado` é reescrito na classe `Gerente`. Caso o programador se esqueça de reescrever o método `getEmpregado()` na subclasse `Gerente` o `Gerente` irá receber a mesma bonificação que um `Empregado`.

Às vezes um comportamento (função do método) é comum em múltiplas subclasses, mas não existe um conjunto padrão de instruções (funcionalidades de um método) que possam ser utilizadas para executar esse comportamento.

Existe uma diferença entre **funções** e **funcionalidades** de um método. A **função** é o **que** o método deve fazer e a **funcionalidade** é **como** ele deve fazer (é o conjunto de instruções empregadas para que o comportamento seja executado).

3.8.1 O poder da abstração

A abstração é a técnica utilizada pelo programador de uma superclasse para forçar os programadores das subclasses a definirem a funcionalidade de um método.

No exemplo anterior vimos que o programador da classe Gerente redefiniu o método (função) `getBonificacao()` definida previamente na classe Empregado, porque esse método não era apropriado para um Gerente.

Esquecer de redefinir um método é comum em um grande sistema com muitos métodos.

Uma **classe abstrata** é uma classe que não permite instâncias; ou seja, não é possível declarar um objeto do tipo da classe abstrata. A classe abstrata deve ser superclasse de uma subclasse para que seus atributos e métodos (comportamentos) sejam utilizados na aplicação. Uma classe abstrata pode conter métodos membros ou métodos operacionais e suas funcionalidades será utilizadas por todas as classes herdeiras. Os métodos abstratos de uma classe abstrata devem ser reescritos na classe filha.

Para definir uma classe como abstrata, devemos colocar a palavra **abstract** antes da palavra `class`. Vejamos no exemplo abaixo:

E

```
//definindo uma classe como abstrata  
abstract class Empregado{}
```

O código acima define que a classe `Empregado` é uma classe abstrata.

Método abstrato não exige funcionalidades (corpo) porque essas serão fornecidas no método redefinido na subclasse. O método abstrato deve ser definido na superclasse sem corpo.

Para definir um método como abstrato devemos escrever:

E

```
modificador abstract tipoDeRetorno nomeDoMetodo(argumentos);
```

Vejamos um exemplo:



```
//definindo o método getBonificacao() como abstrato  
public abstract double getBonificacao();
```

Observe que a palavra **abstract** antes do tipo de retorno do método e o **ponto e vírgula** depois dos parênteses.

O ponto e vírgula é utilizado logo depois dos parênteses para dizer que o método foi definido e que não tem corpo.

Para definirmos a classe *Empregado* como abstrata temos que fazer:



```
abstract class Empregado{  
    public int codigo;  
    double salario;  
    //método abstrato  
    abstract double getBonificacao();  
    //método operacional  
    void setSalario(double s){  
        this.salario = s;  
    }  
}
```

Observe que a classe *Empregado* tem o método *getBonificacao()* como abstrato.

Agora quando o programador da classe *Gerente* herdar da classe *Empregado* ele terá que (forçadamente) implementar (escrever as funcionalidades) do método abstrato *getBonificacao()* da classe *Empregado*. Caso o programador da classe *Gerente* que herda de *Empregado* se esqueça de implementar o método abstrato *getBonificacao()* da classe *Empregado* acontecerá um erro de compilação.




Podemos ver que não faz sentido instanciar uma classe abstrata pois o objeto gerado (se fosse possível) teria um método (função) sem funcionalidade específica tornando o objeto inconsistente.

A abstração é uma forma de "lembrar" o programador que um método está faltando.


Na classe abstrata é possível definir atributos e métodos. Nem todos os métodos precisam ser designados como métodos abstratos pois isso depende da necessidade da classe. O método que for definido como método abstrato obrigatoriamente terá que ser implementado na subclasse.

Os métodos que não forem designados como abstratos não precisarão ser redefinidos na subclasse.


Observações importantes:

	<ul style="list-style-type: none">• Um método abstrato não pode ser acessado diretamente.• Um método abstrato tem que ser redefinido em uma subclasse.• Um método abstrato será acessado depois de redefinido na subclasse.
---	---

Exemplo:

	<pre>abstract class Empregado{ int codigo; float salario; abstract double getEmpregado(); }</pre>
---	---

A classe Gerente que implementa a classe Empregado deve ser escrita da seguinte forma:

	<pre>class Gerente extends Empregado{ int senha; public boolean autentica(int senha){ if(this.senha == senha){ return true; } else{ return false; } //reescrevendo o método abstrato getBonificacao() public double getBonificacao(){ return this.salario * 0.15; } } }</pre>
---	---

Observe que os atributos codigo e salario serão herdados normalmente pela classe Gerente.

No exemplo acima, a classe Empregado é abstrata pois ela possui o método getBonificacao() que é abstrato. **Se uma classe possuir pelo menos um método abstrato ela terá que ser definida como abstrata.**

3.9 Exercícios

1. Defina classe abstrata.
2. Qual a diferença entre função e funcionalidade. Dê um exemplo
3. O que é reescrita de métodos? Dê um exemplo.
4. Qual a sintaxe para se declarar uma classe abstrata.
5. O que acontece quando uma classe abstrata é instanciada?
6. Uma classe que possui um método abstrato tem que ser declarada como abstrata? Justifique sua resposta.
7. Como podemos instanciar objetos de uma classe que possui métodos abstratos?

No computador

1. Implemente uma classe abstrata de nome Forma onde são declarados dois métodos abstratos: float calcularArea(); float calcularPerimetro();
2. Crie, como subclasse de Forma, uma classe de nome Retangulo cujas instâncias são caracterizadas pelos atributos lado e altura ambos do tipo float. Implemente na classe Retangulo os métodos herdados de Forma e outros que ache necessários.
3. Crie, como subclasse de Forma, uma classe de nome Circulo cujas instâncias são caracterizadas pelo atributo raio do tipo float. Implemente na classe Circulo os métodos herdados de Forma e outros que ache necessários.
4. Crie, como subclasse de Retangulo, uma classe de nome Quadrado cujas instâncias são caracterizadas por terem os atributos lado e altura com o mesmo valor.
5. Elabore um programa de teste onde é declarado um array, de dimensão 5, do tipo estático Forma. Nesse array devem ser guardadas instâncias de Retangulo, Circulo e Quadrado seguindo uma ordem aleatória. Nota: para gerar números aleatórios crie primeiro uma instância da classe Random (presente na biblioteca java.util) e para extrair um inteiro entre 0 e n efetue a invocação nextInt(n). Depois implemente um ciclo que percorra o array invocando, relativamente a cada um dos objetos guardados, os métodos calcularArea e calcularPerimetro.

UNIDADE IV

4 Encapsulamento

Ao final da aula o aluno será capaz de:

- Controlar o acesso aos atributos e métodos da classe através dos modificadores `public` e `private`;
- Escrever métodos de acesso a atributos;
- Escrever construtores para as classes;
- Utilizar variáveis e métodos estáticos.

4.1 Modificadores de acesso

Controles de acesso

Em Java temos três modificadores de acesso, sendo eles:

- `public` - modificador de acesso público.
- `protected` - modificador de acesso intermediário entre o modificador `public` e o modificador `private`.
- `private` - modificador de acesso privado.

4.1.1 Modificador `public`

O modificador **`public`** disponibiliza acesso aos atributos e métodos de uma classe para todas as classes do sistema. Com ele é possível fazer um **acesso direto** - utilizando a variável de referência um ponto e o nome do atributo - a um atributo ou método da classe. Vejamos:



```
class Empregado{  
    public String nome;  
    public double salario;  
}
```

No exemplo acima temos os atributos **`codigo`** e **`salario`** como públicos, isso significa que podemos acessá-los diretamente. Vejamos:




```
public class AcessoPublico{  
    public static void main (String[] args){  
        //instancia um objeto Empregado utilizando
```

	<pre>//f como variável de referência. Empregado f = new Empregado(); //Acessando diretamente o atributo nome f.nome = "Gustavo"; //fazendo acesso direto ao atributo salario; f.salario = 8945.3; } }</pre>
--	---

Veja que no código acima alteramos diretamente o valor do atributo salário utilizando a variável de referência **f**. Não foi feita nenhuma verificação para saber se o valor atribuído ao salário está correta.


Poderíamos atribuir ao atributo salário um valor negativo o que não está correto para um atributo salário. Vejamos

	<pre>public class AcessoPublico2{ public static void main(String[] args){ Empregado f = new Empregado(); //atribuindo um valor negativo para salario f.salario = -8945.3; } }</pre>
--	---

O código acima irá compilar sem nenhum problema, mas temos aqui um caso de uma classe mal projetada. O usuário da classe **Empregado** terá que validar o valor do atributo **salario** no programa principal toda vez que for atribuir um valor ao atributo **salario**. Isso não é bom.


4.1.2 Modificador **private**

Para resolver esse problema, podemos utilizar um modificador de acesso (**private**) que não deixa o usuário acessar diretamente o atributo salário dentro da classe. Para atribuir um valor ao atributo, que agora é privado, é necessária a criação de um método de acesso que terá como corpo (funcionalidade) um programa que verifica o valor atribuído, caso o valor atribuído seja negativo ele atribui zero caso contrário ele atribui o valor ao atributo **salario**. Vejamos:

	<pre>class Empregado2{ public String nome; private double salario; //adiciona o método para atribuir o valor a salario public void setSalario(double sal){</pre>
---	---


	<pre> if(sal > 0){ this.salario = sal; } else{ this.salario = 0; } } }</pre>
--	---

No código acima foi implementado o método `public void setSalario(double sal)`. Esse método é `public` e poderá ser acessado por qualquer classe, `void` porque não possui nenhum tipo de retorno (é um método para atribuição) e possui um atributo do tipo `double` (mesmo tipo de salário) chamado `sal`. O parâmetro **sal** só será atribuído a salário se ele for maior que zero, caso contrário zero será atribuído ao atributo `salario`. Vejamos como utilizar isso.

	<pre>public class AcessoPublico3{ public static void main(String[] args){ Empregado2 f = new Empregado2(); //atribui um valor ao atributo salario f.setSalario(8945.3); } }</pre>
--	---

Nesse caso o valor 8945.3 foi passado como parâmetro para o método **setSalario(double sal)**. Dentro do método foi feita a validação do parâmetro e nesse caso ele será atribuído ao atributo `salario` sendo ele positivo e maior que zero.

Quando temos um atributo **private**, para poder acessá-lo só é possível utilizando um método e o acesso direto não é mais permitido. Vejamos:

	<pre>public class AcessoPublico4{ public static void main(String[] args){ Empregado2 f = new Empregado2(); //fazendo um acesso direto ao atributo salario f.salario = 8945.3; } }</pre>
---	---


O código acima gera um erro de compilação pois estamos tentando acessar diretamente um atributo que tem acesso **privado**.

4.1.3 Modificador de acesso - Protected


Nos módulos anteriores vimos os modificadores de acesso **public** em que qualquer classe (variável de referência) pode acessar os atributos e métodos de outra classe e **private** em que somente um método da **própria** classe tem acesso aos seus atributos e métodos.

Para acessarmos os atributos da classe Empregado podemos deixá-los com o modificador de acesso **public**, mas dessa maneira qualquer classe poderia acessá-lo, se os configurarmos para **private** somente métodos da classe Funcionário teria acesso aos atributos. Para que os atributos da classe Empregado sejam acessados somente por ela, pela classe Gerente e pelas classes do mesmo **pacote** (pacote é uma pasta com vários arquivos .class) podemos utilizar o modificador de acesso **protected**.


Nos exemplos abaixo a primeira linha de cada fragmento de código possui a instrução **package** e o nome de um pacote. Vejamos:

	<pre>package pacote1; public class MinhaClasse1{ public static int x; }</pre>
--	---

A classe MinhaClasse1 fica dentro de um pacote chamado **pacote1** que é uma pasta no sistema de arquivos do Windows e a classe MinhaClasse2 fica em um outro pacote chamado pacote2. Vejamos:

	<pre>package pacote2; import pacote1.*; public class MinhaClasse2{ MinhaClasse2(){ MinhaClasse1.x = 30; } }</pre>
---	---


A classe acima utiliza o comando **import** para importar os arquivos do pacote1 (todos os arquivos é representado por *). Para compilar os exemplos acima devemos utilizar o comando **javac -d . nome_da_classe.java** (no console do Windows). Esse comando fará com que seja criada uma pasta que possui o mesmo nome que segue a instrução package e os arquivos .class ficarão dentro dessa pasta que é também conhecida como **pacote**. Ao compilarmos os programa acima, nenhuma mensagem de erro será exibida pois o atributo x da classe MinhaClasse1 que está sendo acessado na classe MinhaClasse2 tem um modificador **public**. Se alterarmos o modificador da classe MinhaClasse1 para **private** teremos:

	<pre>package pacote1; public class MinhaClasse1{ private static int x; } package pacote2; import pacote1.*; public class MinhaClasse2{ MinhaClasse2(){ MinhaClasse1.x = 30; } }</pre>
---	---

O seguinte erro de compilação será mostrado:

```
F:\Curso na WEB\Codigos fonte>javac -d . MinhaClasse2.java
MinhaClasse2.java:7: x has private access in
pacote1.MinhaClasse1
    MinhaClasse1.x = 30;
    1 error
```

Vamos mudar o modificador do atributo x da classe MinhaClasse1 para **protected**.
Vejam os

	<pre>package pacote1; public class MinhaClasse1{ protected static int x; } package pacote2; import pacote1.*; public class MinhaClasse2{ MinhaClasse2(){ MinhaClasse1.x = 30; } }</pre>
---	---


Ao compilarmos o programa o seguinte erro de compilação será mostrado:

```
F:\Curso na WEB\Codigos fonte>javac -d . MinhaClasse2.java
MinhaClasse2.java:7:
x has protected access in pacote1.MinhaClasse1
    MinhaClasse1.x = 30;
    1 error
```

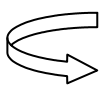
O quadro abaixo mostra os critérios de acessibilidade

Modificador	Mesma classe	Mesmo pacote	Subclasse	Universo
private	sim			
default	sim	sim		
protected	sim	sim	sim	
public	sim	sim	sim	sim

Para acessarmos o atributo x da classe MinhaClasse1 que está no pacote1 através da classe MinhaClasse2 que está no pacote2, teremos que criar um método na classe MinhaClasse1 que dará acesso ao atributo x da classe MinhaClasse1. Este método como o atributo devem ser estáticos (static), pois estão em pacotes diferentes. Vejamos




```
package pacote1;
public class MinhaClasse1{
    protected static int x;
    public static void setX(){
        MinhaClasse1.x = 90;
    }
}
```



Segundo JavaBeans a nomenclatura de métodos é definida com **set** ou **get**. Usamos o pré-fixo **set** quando queremos escrever métodos que irão atribuir valores ou configurar atributos. Usamos o pré-fixo **get** quando queremos escrever métodos que retornam um valor.

4.1.4 static


A palavra **static** cria uma **variável única** compartilhada por todos os objetos **dessa** classe. Dessa forma, quando mudamos seu conteúdo através de um objeto, outro objeto “enxergaria” o mesmo valor. Quando declaramos um atributo como **static**, ele passa a não ser mais um atributo de cada objeto e sim um atributo da classe. A informação fica guardada pela classe e não mais individual para cada objeto. Vejamos:



```
class Empregado{
    String nome;
    double salario;
    static int totalEmpregado;
    //construtor
```


	<pre> Empregado() { this.totalEmpregado += 1; } </pre>
--	--

No exemplo acima, `totalEmpregado` é um atributo estático. O construtor adicionará 1 ao atributo `totalEmpregado` toda vez que o construtor `Empregado()` for utilizado. Esse é um exemplo de atributos estáticos. Podemos ter também métodos que são estáticos. Vejamos:

	<pre> class Empregado{ String nome; double salario; static int totalEmpregado; static void adicionaEmpregado(){ Empregado.totalEmpregado += 1; } } </pre>
---	---

O exemplo acima mostra um método estático que acessa um atributo estático.

Métodos estáticos:

	<ul style="list-style-type: none"> • não utilizam a palavra this para referenciar a atributos. • Métodos estáticos só acessam atributos estáticos. • Acessam atributos estáticos usando o nome da classe o separador ponto e o nome do atributo estático.
---	--

Vejamos:

```
Empregado.totalEmpregado += 1;
```

Métodos e atributos estáticos só podem acessar outros métodos e atributos estáticos da mesma classe, o que faz sentido, já que **dentro de um método estático** não temos acesso a referência **this**, pois um método estático é chamado através da classe e não de um objeto.

4.2 Exercícios

1. Pela convenção, qual o prefixo utilizado em métodos que retornam boolean?
2. Quais as principais características de um construtor?
3. Qual a sintaxe para acessar um atributo estático dentro de um método estático
4. Escreva um exemplo que utilize um método estático
5. Qual a função do modificador public
6. Escreva um exemplo (classe) que utiliza o modificador private e public
7. Qual a função de um método que possui get como prefixo no nome?
8. Escreva um programa que utilize um método get.
9. Qual a função do modificador private?
10. Escreva um exemplo que utilize o modificador private
11. O que é um atributo static?
12. O que acontece, em java, quando criamos uma classe e não definimos um construtor para ela?
13. Qual a função de um método que possui set como prefixo no nome?
14. Quais são os três tipos de modificadores de acesso?
15. O que é um "acesso direto"?
16. O que é um construtor?
17. O que acontece com o construtor padrão quando criamos um construtor na classe?
18. Atributos são inicializados automaticamente. Qual o valor inicial para tipos numéricos primitivos, String, objetos e boolean?

No computador

1. Escreva uma classe chamada Pessoa que possua nome e e-mail como atributos privados. Crie os métodos para configurar e mostrar os valores dos atributos.
2. Escreva uma classe que se chama Aluno que herda de Pessoa. Adicione ao aluno um atributo privado chamado série, um atributo privado do tipo boolean chamado aprovado, os atributos privados chamados nota1, nota2, nota3 e nota4. O método para configurar a série deverá aceitar somente os valores de 1 a 8. Caso o valor não esteja nessa faixa, zero deverá ser atribuído ao atributo série. O método para configurar os atributos nota1, nota2, nota3 e nota4 deverão aceitar somente números positivos de 1 a 10 caso o número atribuído não esteja nessa faixa, zero deverá ser atribuído ao atributo. O método para atribuir o valor ao atributo aprovado

dependerá das notas atribuídas aos atributos notas utilizando o cálculo: $(n1+n2+n3+n4)/4 \geq 6$. Caso a média seja maior ou igual a 7 o atributo receberá verdadeiro (true) caso contrário receberá falso (false).

3. Crie uma classe chamada AlunoFaculdade que herda de Aluno e reescreva o método aprovado de forma que a média mude de 6 para 8. Reescreva o método série para que o aluno se enquadre na faixa de 1 a 4.
4. Construa uma aplicação para executar as classes acima.
5. (Estudo dirigido)
Vamos construir uma janela gráfica em Java, rever alguns conceitos e aprender novos conceitos.

Vamos criar uma classe que configure uma janela gráfica para isso precisamos importar o pacote `javax.swing.*` (pacotes foram descritos na primeira aula). Vejamos:

```
//importando os pacotes de java (bibliotecas)
import javax.swing.*;
public class Janela extends JFrame{
    //construtor
    Janela(){
        //método que configura o tamanho da janela
        setSize(300,200);
        //método que configura a localização da janela
        setLocation(10,20);
        //método que configura o título da janela
        setTitle("Minha primeira janela");
        //método que mostra a janela
        setVisible(true);
    }
}
```

Podemos identificar nessa classe a **biblioteca** ou **pacote** que foi importada (`javax.swing.*`) essa linha nos diz que existem classes, representadas por *, dentro de um pacote chamado swing que está dentro de outro pacote chamado javax. Entenda pacotes como se fossem pastas (diretórios) que serão usadas para a criação da janela. Podemos também identificar os métodos que são iniciados por **set** que são `setSize`, `setLocation`, `setTitle`, `setVisible`, e todos eles possuem parâmetros. que são a largura e altura da janela - `setSize(300,200)` - posição da janela na tela pelas coordenadas x e y - `setLocation(10,20)` - o título da janela - `setTitle("Minha primeira janela")` - e a visibilidade da janela - `setVisible(true)`.

O código abaixo mostra como instanciar a classe acima e torná-la executável. A classe abaixo pode ser chamada de classe executável ou classe programa. Vejamos:

```
public class MostraJanela {
```

```
public static void main(String[] args){  
    Janela minhaJanela = new Janela();  
}  
}
```

A classe acima mostra uma classe chamada MostraJanela e essa classe **tem um** objeto Janela que será instanciado e referenciado pela variável de referência minhaJanela. Como todo o código de construção da janela está no construtor de Janela, então a janela será construída automaticamente.

6. (Estudo dirigido)

Escreva uma classe chamada Cliente que possua os atributos privados nome, endereço e cpf do tipo String, idade do tipo int. Sua classe deverá possuir o método setCPF que tem como argumento o cpf. O corpo do método setCPF deve chamar um método chamado validaCPF que tem como argumento o cpf e depois de validado deve atribuir o valor do cpf ao atributo cpf. A classe também deverá ter um método chamado validaCPF que tem como argumento o cpf. O corpo do método validaCPF deve verificar se os caracteres digitados são somente números. Use o método charAt(posição da string) da classe String para ver o valor de cada posição da string.

```
public class Cliente{  
    private String nome, endereco, cpf;  
    private int idade;  
  
    String validaCPF(String cpf){  
        String dominio = "0123456789";  
        int cont = 0;  
        for(int i=0; i<11; i++){  
            for(int j=0; j<10; j++){  
                if(cpf.charAt(i) == dominio.charAt(j)){  
                    cont++;  
                }  
            }  
        }  
        if(cont == 11){  
            this.cpf = cpf;  
            return cpf;  
        }  
        else  
        {  
            return "Erro!";  
        }  
    }  
  
    void setCPF(String cpf){  
        this.cpf = validaCPF(cpf);  
    }  
}
```

```
    }

    String getCPF() {
        return this.cpf;
    }

    void setNome(String nome) {
        this.nome = nome;
    }

    void setEndereco(String end) {
        this.endereco = end;
    }

    void setIdade(int i) {
        if (i > 0) {
            this.idade = i;
        }
    }

    String getNome() {
        return this.nome;
    }

    String getEndereco() {
        return this.endereco;
    }

    int getIdade() {
        return this.idade;
    }
}
```



O fragmento de código acima que está em *itálico* pode ser substituído por um método chamado **decode** da classe **Integer**. O método **decode(String x)** tem como parâmetro uma *String* e devolve essa *String* se ela contiver apenas números. Caso contrário ela devolve uma exceção (Exception). O código abaixo mostra o código acima utilizando o método **decode**.

```
public class Cliente {
    private String nome, endereco, cpf;
    private int idade;

    String validaCPF(String cpf){

        try{
```

```
        float x = Float.parseFloat(cpf);
        return ""+cpf;
    }

    catch(NumberFormatException nfe){
        return "Existe caracteres não numéricos na string";
    }
}

void setCPF(String cpf){
    this.cpf = validaCPF(cpf);
}

String getCPF(){
    return this.cpf;
}

void setNome(String nome){
    this.nome = nome;
}

void setEndereco(String end){
    this.endereco = end;
}

void setIdade(int i){
    if (i > 0){
        this.idade = i;
    }
}

String getNome(){
    return this.nome;
}

String getEndereco(){
    return this.endereco;
}

int getIdade(){
    return this.idade;
}
}
```

Observação importante: Ao utilizarmos os métodos que já existem na linguagem vemos que o trabalho diminui sensivelmente. Conhecer as classes

e seus métodos é fundamental para um bom desenvolvimento em Orientação a Objetos.

7. Crie uma classe chamada Conta que possui os seguintes atributos privados saldo, limite do tipo double e dono do tipo Cliente. Construa os getters e setters para os todos atributos. Crie um programa para testar sua classe.
8. Na classe Conta adicione um construtor que quando utilizado mostra a mensagem "Construindo uma conta" na tela.
9. Faça com que seu construtor além de mostrar a mensagem na tela, acumule em um atributo totalContas a quantidade de conta abertas.

```
public class Conta{
    private double saldo, limite;
    private Cliente dono;
    static int totalconta;

    public Conta(){
        System.out.println("Construindo uma conta");
        totalconta++;
    }

    public void setSaldo(double saldo){
        this.saldo = saldo;
    }

    public double getSaldo(){
        return this.saldo;
    }

    public void setLimite(double limite){
        this.limite = limite;
    }

    public double getLimite(){
        return this.limite;
    }
}

public class ExecutaCliente{
    public static void main(String[] args){
        Cliente c = new Cliente();
        c.setCPF("92695787934");
        System.out.println(c.getCPF());
        Conta conta1 = new Conta();
        Conta conta2 = new Conta();
    }
}
```

```
        conta1.setSaldo(8990);  
        System.out.println(conta1.getSaldo());  
        System.out.println(conta1.totalconta);  
    }  
}
```

Exercícios complementares

1. Crie a classe Empregado com os atributos private salario (float), public nome (string).
2. Crie uma classe chamada MeuEmpregado, instancie um funcionário qualquer e tente modificar ou ler o salário. O que acontece?
3. Crie os getters e setters necessários da sua classe Empregado
4. Modifique a sua classe MeuEmpregado que acessam e modificam atributos de um Empregado para utilizar getters e setters.
5. Adicione um atributo na classe Empregado do tipo int que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo Empregado. O primeiro Empregado instanciado tem identificador 1, o segundo 2 e assim por diante.
6. Crie um getter para o identificador. Você deve criar um setter? Justifique.
7. Por que o código não compila?

```
class Teste{  
    int x = 45;  
    public static void main(String[] args){  
        System.out.println(x);  
    }  
}
```

Como devemos resolver esse problema?

UNIDADE V

Ao final da aula o aluno será capaz de:

- Entender as interfaces;
- Declarar interfaces;
- Utilizar interfaces;
- Entender a vantagem do polimorfismo;
- Escrever métodos polimórficos;
- Utilizar métodos polimórficos;
- Diferenciar o relacionamento do tipo "tem um" e o relacionamento "é um" (composição).
- Criar classes utilizando composição.

5 Interface

Uma interface Java é muito parecida com uma classe "abstrata pura" (somente com os métodos sem funcionalidades). Na interface **todos** os métodos são implicitamente **abstract** e **public**, e todos os atributos são implicitamente **static** e **final**.

A interface possui sua sintaxe para a declaração muito parecida com a sintaxe de declaração de classe, porém seu corpo define apenas os métodos com ou sem assinaturas e constantes.

Interface e classe abstrata são parecidas, uma das diferenças é que a classe abstrata pode ter métodos concretos implementados e a interface não.

Quando declaramos uma interface pública, fazemos um contrato entre o código do cliente e a classe que implementa essa interface. Esse contrato diz que devemos implementar todos os métodos dessa interface.

Exemplo de uma classe Java:



```
public interface BancoDeDados{  
    public void inserir();  
    public void excluir();  
}
```



```
public class Escola implements BancoDeDados{  
    public void inserir(){  
        //escreve o código para inserção;  
    }  
}
```

```
public void excluir(){  
    //escreve o código para excluir;  
}  
}
```

5.1 Interfaces são usadas para:

- Declarar métodos que podem ser usados e devem ser implementados de formas diferentes em classes diferentes com propósitos semelhantes.
- Identificar as similaridades entre classes não relacionadas sem ter um relacionamento de classes.
- Simular a herança múltipla (que existe em C++) declarando uma classe que implementa diversas interfaces.



Podemos declarar variáveis dentro de uma interface. Esta variável será automaticamente **public**, **static** e **final**.

5.2 Palavra chave final

A palavra-chave final pode ser usada com declarações de classes, métodos ou atributos. Vejamos:



```
public final class Pagina { // não poderá ter subclasses  
    public final int SIZE = 15; //valor constante  
    // este método nunca poderá ser sobrescrito  
    public final void tenteSobrescrever(){ }  
}
```

5.2.1 Uma classe final

- Nunca poderá ser herdada.
- A palavra-chave final pode também ser aplicada a métodos e a atributos de uma classe.
- Um método final não pode ser redefinido em classes derivadas (segurança).
- Um atributo final não pode ter seu valor modificado, ou seja, define valores constantes.
- Apenas valores de tipos primitivos podem ser utilizados para definir constantes.
- O valor do atributo deve ser definido no momento da declaração, pois não é permitida nenhuma atribuição em outro momento.
- Um objeto final não consegue instanciar outro objeto.

Em Java

Um bom exemplo do uso de interfaces é adicionar funcionalidade a um botão. Vejamos o exemplo:



```
//cria uma janela
import javax.swing.*;
import java.awt.event.*;
public class Janela{
    JButton botao = new JButton("Pressione!");
    JFrame minhaJanela = new JFrame("Janela Teste");
    Janela(){
        minhaJanela.setLayout(null);
        botao.setBounds(10, 10, 150, 25);
        TrataBotao tb = new TrataBotao();
        botao.addActionListener(tb);
        minhaJanela.add(botao);
        minhaJanela.setSize(300,200);
        minhaJanela.setLocation(10,20);
        minhaJanela.setVisible(true);
    }
    //classe Interna
    private class TrataBotao implements ActionListener{
        //precessa eventos capturados
        public void actionPerformed(ActionEvent e){
            if(e.getSource()== botao){

                JOptionPane.showMessageDialog(minhaJanela,"Botao
                pressionado");
                System.exit(0);
            }
        }
    }
}
```

O código acima mostra a utilização de interfaces em programação orientada a objetos. Vamos analisar o código:

- Foi importado o pacote **swing** para construção da janela e o pacote **event** para tratamento de eventos.
- Foram criados **dois objetos** sendo um botao do tipo JButton e minhaJanela do tipo JFrame.
- O construtor da classe - Janela() - possui como funcionalidades:

- Configurar o layout da janela;
- Configurar a posição do botão;
- Adicionar ao botão um "ouvidor de eventos" `addActionListener`;
- Adicionar o botão à janela;
- Configurar o tamanho da janela;
- Configurar a posição da janela na tela;
- Mostrar a janela na tela.

A classe interna chamada `TrataBotao` foi criada dentro da classe `Janela` (por isso classe interna) e implementa a interface **`ActionListener`**. O método que foi implementado é o **`actionPerformed`** que possui como parâmetro um **objeto** do tipo **`ActionEvent`**. A classe `ActionEvent` possui o método **`getSource()`** que identifica a origem do evento. Como essa identificação da origem do evento está dentro de uma estrutura condicional, caso o botão pressionado seja igual ao registrado então um evento será gerado e nesse caso será mostrada uma janela com a mensagem "Botao pressionado" e depois a janela será fechada.

Uma observação quanto aos parâmetros do método `showMessageDialog` da classe `JOptionPane` é que o primeiro argumento não é `null`, e sim o objeto `minhaJanela` fazendo com que a caixa de diálogo seja aberta no centro da janela (chamada de janela pai). Quando você utiliza o parâmetro `null`, java entende que a janela pai é todo o monitor então abre a janela de diálogo no centro da tela.

5.2.2 Exercícios

1. Escreva o código que exemplifique uma classe final.
2. O que acontece com um método quando ele é declarado como final?
3. Qual a diferença de sintaxe de uma classe abstrata e uma interface?
4. O que acontece com o compilador ao "enxergar" a palavra **`implements`** no código de uma classe?
5. Por que de usar interface?
6. O que acontece com uma classe quando ela é declarada como final?
7. Na interface todos os métodos são implicitamente `abstract` e `public` e os atributos são implicitamente `static` e `final`. Explique cada um dos modificadores citados na afirmação acima.
8. Qual a sintaxe para a declaração de uma interface?


No computador

1. Escreva uma classe concreta chamada `Conta` que possua os atributos privados do tipo `double` `saldo`, `débito` e `crédito`.
2. Crie um método chamado `deposito` que possua um parâmetro chamado `valorDepositado` do tipo `double` em que um usuário poderá fazer um depósito de uma determinada quantia. Um outro método chamado `retirada` deverá ser


- criado tendo como parâmetro um valorRetirado. Os dois métodos deverão afetar o atributo saldo.
3. Escreva um método chamado extrato para mostrar o valor do saldo atual.
 4. Escreva uma classe que irá instanciar dois objetos de Conta um chamado c1 e outro chamado c2. Faça com que o cliente1 (c1) deposite 100.45 na conta e que o cliente c2 deposite 400.0 na mesma conta. Depois mostre o extrato da conta. Faça o cliente c3 retirar 20.0 da conta e mostre o extrato novamente.
 5. Escreva uma interface chamada Banco que possui o método double relatório(). O método relatório é utilizado pelas classes que o implementam para mostrar dados da conta. Essa interface tem um atributo chamado double limite com um valor inicial de 100.00
 6. Na classe Conta o relatório deverá se implementado da seguinte forma: ter uma variável local do tipo double chamada saldoAtual que receberá o limite mais o saldo e imprimirá na tela o saldoAtual.
 7. Faça as seguintes modificações no exercício anterior:
 - Não coloque o saldo como static, compile e execute o programa. Anote a mensagem de erro.
 8. Escreva um programa que desenhe uma janela com um botão no centro do monitor. Ao ser pressionado o botão, seu nome deverá ser mostrado na tela em uma janela de diálogo.

5.3 Polimorfismo

Na herança vimos que o Gerente **é um** Empregado, pois ele é uma extensão deste. Polimorfismo (Poli = várias + morfos = formas) é a capacidade de um objeto poder ser referenciado de várias formas. Dada a classe:

	<pre>class Gerente extends Empregado{ int senha; public boolean autentica(int senha){ if(this.senha == senha){ return true; } else{ return false; } } //reescrevendo o método getBonificacao() public double getBonificacao(){ return this.salario * 0.15; } }</pre>
---	--

Se instanciarmos um objeto da seguinte forma:

	<pre>class TestaGerente{ public static void main (String[] args){ //instanciando um objeto Empregado (nesse caso gerente) Empregado g = new Gerente(); //utilizando os atributos e métodos do objeto Empregado g.salario = 5000; //Decide o método que será usado na execução System.out.println("Gerente: "+g.getBonificacao()); //instanciando um objeto Empregado Empregado f = new Empregado(); //utilizando os atributos e métodos do objeto Empregado f.salario = 5000; System.out.println("Empregado: "+f.getBonificacao()); } }</pre>
---	---


Qual será a saída desse programa?


```
F:\Curso na WEB\Codigos fonte>java TestaGerente
Gerente: 750.0
Empregado: 500.0
```


No Java a chamada de método sempre vai ser decidida em tempo de execução "**late binding**". O Java procura o objeto na memória e aí sim decide qual método vai ser chamado, sempre relacionando com sua classe "de verdade", e não a que estamos usando para referenciá-lo. Apesar de estarmos nos referenciando a **Gerente** como um **Empregado**, o método executado é o do **Gerente**.

5.3.1 Por que usar polimorfismo?

Polimorfismo é muito utilizado quando temos um método que recebe um argumento do tipo de outro objeto. No exemplo abaixo temos um método que recebe com argumento um tipo **Funcionário**. Vejamos

	<pre>class ControleBonificacao{ private double totalDeBonificacoes; public void registra(Empregado f){ this.totalDeBonificacoes += f.getBonificacao(); } public double getTotalDeBonificacoes(){ return this.totalDeBonificacoes; } }</pre>
---	---

	<pre>public class TestaControleBonificacao{ public static void main(String[] args){ ControleBonificacao c = new ControleBonificacao(); Gerente g = new Gerente(); g.salario=5000; c.registra(g); Empregado f = new Empregado(); f.salario = 5000; c.registra(f); System.out.println("Total bonificações:"+c.getTotalDeBonificacoes()); } }</pre>
---	--

A saída do programa é:

```
F:\Curso na WEB\Codigos fonte>java TestaControleBonificacao
Total de bonificações: 1250.0
```

A classe **ControleBonificacao** soma o total de bonificações pagos pela empresa a seus empregados. O método **registra(Empregado f)** tem como argumento um tipo


Empregado. Ao ser passado como parâmetro um tipo de funcionário **Gerente** o método **getBonificacao()** da classe Gerente é invocado **automaticamente** e retorna o valor de 75. Ao ser passado como parâmetro um tipo de funcionário **Empregado** o método **getBonificacao()** da classe Empregado é invocado automaticamente e retorna o valor 50 que é adicionado ao valor de 75 retornado pelo método **getBonificacao()** da classe Gerente perfazendo o total de 125. Essa invocação de método é chamada de **Ligação tardia ou late binding** porque é o Java que decide qual método chamar e não o programador.

Não importa que dentro do método **registra(Empregado f)** da classe ControleBonificacao tenha o tipo **Empregado** como argumento, quando ele receber um parâmetro **Gerente**, o método **getBonificacao()** da classe Gerente é que será invocado.

Caso criemos uma classe Secretaria que seja subclasse da classe Empregado, precisaremos mudar a classe ControleBonificacao? A resposta é **não**. Basta a classe Secretaria reescrever o método **getBonificacao()**. Esse é o poder do **polimorfismo** juntamente com a **herança** e **reescrita** de método, diminuir o acoplamento entre as classes para evitar que novos códigos resultem em modificações em vários lugares.


Além do relacionamento **é um** temos o relacionamento **tem um** que é chamado de **composição**. Suponhamos a classe Funcionário que possui nome e salário. Bem, se o funcionário tem um salário ele deve ter uma conta em um banco. Sendo assim dizemos que o Funcionário **tem uma** Conta. A representação pode ser escrita da seguinte forma.

A classe Conta

	<pre>public class Conta{ int numero; float deposito; float saldo; void setNumero(int n){ this.numero = n; } int getNumero(){ return this.numero; } void setDeposito(float d){ if(d > 0){ this.saldo += d; } else{ deposito = 0; } } }</pre>
---	---

	<pre> } } float getSaldo() { return this.saldo; } }</pre>
--	--

A classe Funcionário ficará da seguinte forma:

	<pre>class Empregado{ String nome; double salario; Conta contaEmpregado; //diz que funcionário tem uma conta. }</pre>
---	--

O atributo contaEmpregado é do tipo conta e pode utilizar todos os métodos que a classe conta disponibiliza.

5.4 Exercícios

1. Crie uma classe chamada Automóvel que possua um construtor que imprime a string “Eu sou um automóvel”. A classe Automóvel deverá possuir um método chamado combustivel que retorna a String “Uso combustivel” e um método chamado numeroRodas que retorna a String “Tenho rodas”
2. Crie uma classe chamada Carro que estende da classe Automóvel que possua um construtor que imprime a string "Eu sou um carro!", um método chamado combustivel que retorna a string "Uso álcool" e um método chamado numeroRodas que retorna a string "Tenho 4 rodas!".
3. Crie uma classe chamada Moto que estende da classe Automóvel que possua um construtor que imprime a string "Eu sou uma moto!", um método chamado combustivel que retorna a string "Uso gasolina!" e um método chamado numeroRodas que retorna a string "Tenho 2 rodas!".
4. Crie uma classe chamada AplicacaoPolimorfismo que possui um método abaixo:

```
public static void imprime(Automovel x){  
    System.out.println(x.combustivel());  
    System.out.println(x.numeroRodas());  
}
```

Essa classe também possui o método public static void main(String[]args) que instancia os objetos:

```
//instanciando um automovel
Automovel a = new Automovel();
System.out.println("-----");
//instanciando um carro
Automovel c = new Carro();
System.out.println("-----");
//instanciando uma moto
Automovel m = new Moto();
System.out.println("-----");
```

Ela deverá também chamar o método imprime da seguinte maneira:

```
//chama o método imprime()
imprime(a);
System.out.println("-----");
imprime(c);
System.out.println("-----");
imprime(m);
```

Execute o exercício acima e escreva suas conclusões sobre os resultados obtidos.

6. Respostas dos exercícios

Exercícios da Unidade III

1. Escreva uma classe chamada ClasseA que possua um método chamado metodoA que retorna "eu sou o método A".

```
package exercicio3_3;

public class ClasseA {
    public String metodoA(){
        return "Eu sou o método A";
    }
}
```

2. Escreva uma classe chamada ClasseB que estende da ClasseA e possua um método chamado metodoB que retorna "eu sou o método B".

```
package exercicio3_3;

public class ClasseB extends ClasseA {
    public String metodoB(){
```

```
        return "Eu sou o método B";  
    }  
}
```

3. Crie uma aplicação, em Java, que instancie um objeto da ClasseB e imprima utilizando o console o retorno dos dois métodos criados.

```
package exercicio3_3;  
  
public class AppHeranca {  
    public static void main(String[] args){  
        ClasseB cb = new ClasseB();  
        System.out.println(cb.metodoA());  
        System.out.println(cb.metodoB());  
    }  
}
```

4. Crie uma classe chamada Animal que possua os atributos peso e cor. O construtor dessa classe deverá atribuir o peso e a cor para o animal no momento da instanciação. Crie os métodos para configurar e ler esses atributos (métodos set e get). Um método chamado mostraAnimal deverá ser criado para mostrar o peso do animal e a cor do animal. O método mostraAnimal deverá escrever na janela gráfica "O animal tem cor " +cor+ "e pesa: " +peso+ " kilos."

```
package exercicio3_3;  
  
public class Animal {  
    float peso;  
    String cor;  
  
    Animal(float p, String c){  
        this.peso = p;  
        this.cor = c;  
    }  
  
    void setPeso(float p){  
        this.peso = p;  
    }  
  
    void setCor(String c){  
        this.cor = c;  
    }  
  
    float getPeso(){  
        return this.peso;  
    }  
}
```

```
String getCor(){
    return this.cor;
}

String mostraAnimal(){
    return "O animal tem a cor "+this.cor+" e psea
"+this.peso+" kilos";
}
}
```

5. Crie uma classe chamada Cachorro que estende a classe Animal e coloque os atributos nome e raça. Crie um construtor para atribuir o nome, raça, peso e cor do animal. No corpo do construtor chame o construtor da superclasse para atribuir peso e cor ao animal instanciado. Crie um outro construtor que atribua somente o nome do animal ao atributos nome no momento em que o objeto for instanciado. Crie métodos para configurar e mostrar os atributos nome e raça.

```
package exercicio3_3;

public class Cachorro extends Animal{
    String nome, raca;

    Cachorro(){}

    Cachorro(String nome, String raca, float peso, String
cor){
        super(peso, cor);
        this.nome = nome;
        this.raca = raca;
    }

    Cachorro(String nome){
        this.nome = nome;
    }

    public void setNome(String n){
        this.nome = n;
    }

    public void setRaca(String raca){
        this.raca = raca;
    }

    public String getNome(){
        return this.nome;
    }
}
```

```
        public String getRaca() {  
            return this.raca;  
        }  
    }  
}
```

6. Crie uma classe chamada Cox que estende de Cachorro. Coloque como atributo tosa do tipo String. Crie um construtor padrão e um construtor que tenha como argumentos o peso, cor e tosa. Os argumentos peso e cor deverão ser passados para o construtor da super classe e tosa deverá ser atribuído ao atributo tosa. Crie um método chamado mostraAnimal que deverá escrever na janela "Este cachorro está tosado!" se tosa a variável tosa estiver a string "foi tosado" e a frase "Você precisa tosar o cachorro!" se a variável tosa estiver a string "não foi tosado". Crie um método que mostre o valor configurado para tosa (getTosa).

```
package exercicio3_3;  
  
public class Cox extends Cachorro {  
    String tosa;  
  
    Cox() {}  
  
    Cox(Float peso, String cor, String tosa) {  
        super(peso, cor);  
        this.tosa = tosa;  
    }  
  
    String getTosa() {  
        return tosa;  
    }  
  
    public String mostraAnimal() {  
        if(this.tosa == "foi tosado") {  
            return "Este cachorro está tosado";  
        } else {  
            return "Você precisa tosar o cachorro";  
        }  
    }  
}
```

"Esse exercício solicita que seja criado um construtor que chama um outro construtor. Veja que na classe superior não existe um método construtor com os argumentos peso e cor conforme foi solicitado no exercício. Para resolver, crie um construtor com os argumentos peso e cor na classe Cachorro."

7. Escreva um programa que tenha uma tela gráfica com os campos Nome, Raça, Peso, Cor e Tosa sendo Tosa um comboBox. A aplicação deverá ter dois botões: Gravar e Mostrar. Ao ser pressionado o botão Gravar os dados Peso, Cor e Tosa deverão ser passados para o construtor da classe Cox. Os valores dos campos Nome e Cor deverão ser passados para o método herdado por Cox da classe Cachorro setName e setRaca. Ao pressionar o botão Mostrar uma janela deverá ser mostrada com todos os dados cadastrados. Os dados deverão ser lidos dos métodos get das classes.

```
package exercicio3_3;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class AppAnimal implements ActionListener{
    //declara os atributos
    private Cox cachorro;
    private JFrame janela, jRelatorio;
    public JLabel lblNome, lblRaca, lblPeso, lblCor, lblTosa;
    private JComboBox cmbTosa;
    private JTextField tfNome, tfRaca, tfPeso, tfCor;
    private String[] opcaoTosa = {"foi tosado", "não foi
tosado"};
    private JButton btoGravar, btoMostrar;

    //construtor da janela
    AppAnimal(){
        //instancia os elementos da tela
        janela = new JFrame("Cadastro do cachorro");
        jRelatorio = new JFrame("Relatório");

        lblNome = new JLabel("Nome: ");
        lblRaca = new JLabel("Raça: ");
        lblPeso = new JLabel("Peso: ");
        lblCor = new JLabel("Cor: ");
        lblTosa = new JLabel("Tosa: ");
        tfNome = new JTextField(20);
        tfRaca = new JTextField(30);
        tfPeso = new JTextField(5);
        tfCor = new JTextField(20);
        cmbTosa = new JComboBox(opcaoTosa);
        btoGravar = new JButton("Gravar");
        btoMostrar = new JButton("Mostrar");

        //configura o leiaute da janela para nulo
```



```
janela.setLayout(null);

//configura o posicionamento dos elementos na janela
lblNome.setBounds(10,10,90,20);
tfNome.setBounds(60,10,120,20);
lblRaca.setBounds(10,40,90,20);
tfRaca.setBounds(60,40,150,20);
lblPeso.setBounds(10, 70, 40, 20);
tfPeso.setBounds(60, 70, 80, 20);
lblCor.setBounds(10, 100, 40, 20);
tfCor.setBounds(60, 100, 150, 20);
lblTosa.setBounds(10, 130, 50, 20);
cmbTosa.setBounds(60, 130, 150, 20);
btoGravar.setBounds(120, 160, 80,30);
btoMostrar.setBounds(210, 160, 80, 30);

//registra eventos para os botões
btoGravar.addActionListener(this);
btoMostrar.addActionListener(this);

//adiciona os elementos na janela
janela.add(lblNome);
janela.add(tfNome);
janela.add(lblRaca);
janela.add(tfRaca);
janela.add(lblPeso);
janela.add(tfPeso);
janela.add(lblCor);
janela.add(tfCor);
janela.add(lblTosa);
janela.add(cmbTosa);
janela.add(btoGravar);
janela.add(btoMostrar);

//configura o tamanho da janela
janela.setSize(400,240);
//configura o posicionamento da janela
janela.setLocation(400,100);
//mostra a janela
janela.setVisible(true);
}

//implementa o método da interface
@SuppressWarnings("static-access")
public void actionPerformed(ActionEvent e){
    if(e.getSource() == btoGravar){
```

```
        cachorro = new
Cox(Float.parseFloat(this.tfPeso.getText()),
this.tfCor.getText(), (String)cmbTosa.getSelectedItem());
        cachorro.setNome(tfNome.getText());
        cachorro.setRaca(tfRaca.getText());
        cachorro.setNome(tfNome.getText());
        cachorro.setRaca(tfRaca.getText());
    }
    if(e.getSource()==btoMostrar){
        //declara os labels para nova janela
        JLabel rNome, rRaca, rPeso, rCor, rTosa;
        //instancia os labels
        rNome = new JLabel(cachorro.getNome());
        rRaca = new JLabel(cachorro.getRaca());
        rPeso = new JLabel("cachorro.getPeso()");
        rCor = new JLabel(cachorro.getCor());
        rTosa = new JLabel(cachorro.getTosa());

        //posiciona os elementos na janela
        lblNome.setBounds(10,10,90,20);
        rNome.setBounds(60,10,120,20);
        lblRaca.setBounds(10,40,90,20);
        rRaca.setBounds(60,40,150,20);
        lblPeso.setBounds(10, 70, 40, 20);
        rPeso.setBounds(60, 70, 80, 20);
        lblCor.setBounds(10, 100, 40, 20);
        rCor.setBounds(60, 100, 150, 20);
        lblTosa.setBounds(10, 130, 50, 20);
        rTosa.setBounds(60, 130, 150, 20);

        //configura o leiaute da janela para nulo
        jRelatorio.setLayout(null);
        jRelatorio.setSize(400,240);
        jRelatorio.setLocation(400,200);
        jRelatorio.add(lblNome);
        jRelatorio.add(rNome);
        jRelatorio.add(rRaca);
        jRelatorio.add(lblRaca);
        jRelatorio.add(rPeso);
        jRelatorio.add(lblPeso);
        jRelatorio.add(rCor);
        jRelatorio.add(lblCor);
        jRelatorio.add(rTosa);
        jRelatorio.add(lblTosa);

        //dispara a janela
        jRelatorio.setVisible(true);
    }
}
```

```
    }  
    }  
}
```

```
//Observe que tivemos que alterar o tipo de retorno do combo  
box tosa para String. Na verdade ele retorna um Object  
//Tivemos que fazer um cast para que o tipo retornado Object  
se transforme em um String. Veja que o campo peso é um tipo  
float e que foi convertido para String quando foi  
//instanciado o label rPeso apenas usando aspas duplas.
```

3.8 Exercícios

1. Qual a vantagem da reescrita de métodos?

Métodos com mesmo nome e assinatura podem ter funcionalidades diferentes. Não é necessário escrever muitos métodos todos com nomes diferentes.

2. O método de qual classe será chamado pelo comando `super.NomeDoMetodo()`?

Será chamado o método da classe mãe ou superclasse.

3. Escreva uma classe chamada Pessoa que possua nome e e-mail como atributos privados. Crie os métodos para configurar e mostrar os valores dos atributos.

```
package exercicio3_3;  
  
public class Pessoa {  
    private String nome, eMail;  
  
    public void setNome(String n){  
        this.nome = n;  
    }  
  
    public void setEmail(String em){  
        this.eMail = em;  
    }  
  
    public String getNome(){  
        return this.nome;  
    }  
  
    public String getEmail(){  
        return this.eMail;  
    }  
}
```

4. Escreva uma classe que se chama Aluno que herda de Pessoa. Adicione ao aluno um atributo privado inteiro chamado série, um atributo privado do tipo boolean chamado aprovado, os atributos privados (float) chamado nota1, nota2, nota3 e nota4. O método para configurar a série deverá aceitar somente os valores de 1 a 8. Caso o valor não esteja nessa faixa, zero deverá ser atribuído ao atributo série. O método para configurar os atributos nota1, nota2, nota3 e nota4 deverão aceitar somente números positivos de 1 a 10 caso o número atribuído não esteja nessa faixa, zero deverá ser atribuído ao atributo. O método para atribuir o valor ao atributo aprovado dependerá das notas atribuídas aos atributos notas utilizando o cálculo: $(n1+n2+n3+n4)/4 \geq 6$. Caso a média seja maior ou igual a 6 o atributo receberá verdadeiro (true) caso contrário receberá falso (false).

```
package exercicio3_3;

package exercicio3_3;

public class Aluno extends Pessoa {
    private int serie;
    private boolean aprovado;
    private float nota1, nota2, nota3, nota4;

    //método para configurar a série
    public void setSerie(int n){
        if (n >= 1 && n <= 8){
            this.serie = n;
        }
        else{
            this.serie = 0;
        }
    }

    //método para configurar a nota1
    public void setNota1(float nota){
        if (nota >= 1 && nota <= 10){
            this.nota1 = nota;
        }else{
            this.nota1 = 0;
        }
    }

    //método para configurar a nota2
    public void setNota2(float nota){
        if (nota >= 1 && nota <= 10){
            this.nota2 = nota;
        }else{
            this.nota2 = 0;
        }
    }
}
```

```
}
//método para configurar a nota3
public void setNota3(float nota){
    if (nota >= 1 && nota <= 10){
        this.nota3 = nota;
    }else{
        this.nota3 = 0;
    }
}
//método para configurar a nota4
public void setNota4(float nota){
    if (nota >= 1 && nota <= 10){
        this.nota4 = nota;
    }else{
        this.nota4 = 0;
    }
}

//método para configurar aprovação
protected void setAprovado(boolean estado){
    this.aprovado = estado;
}

public void setAprovado(){
    if((this.nota1 + this.nota2 + this.nota3 + this.nota4)/4
    >= 6)
        this.aprovado = true;
    else
        this.aprovado = false;
}

//métodos que retornam valores
public float getNota1(){
    return this.nota1;
}

public float getNota2(){
    return this.nota2;
}

public float getNota3(){
    return this.nota3;
}

public float getNota4(){
    return this.nota4;
}
```

```
}  
//método que retorna o status de aprovação do aluno.  
public boolean getAprovado() {  
    return this.aprovado;  
}  
  
}
```

5. Crie uma classe chamada `AlunoFaculdade` que herda de `Aluno` e reescreva o método `aprovado` de forma que a média mude de 6 para 8. Reescreva o método `série` para que o aluno se enquadre na faixa de 1 a 4.

```
package exercicio3_3;  
  
public class AlunoFaculdade extends Aluno {  
    public void setAprovado() {  
        if((getNota1() + getNota2() + getNota3() + getNota4())/4 >=  
8)  
            setAprovado(true);  
        else  
            setAprovado(false);  
    }  
}
```

"Para resolver esse exercícios teremos que ter na classe `Aluno` métodos que retornem os valores das notas.

Se as notas são privadas em `Aluno` elas só podem ser acessadas via métodos da própria classe `Aluno`.

Caso você coloque as notas como `public` então não justifica escrever métodos para atribuir ou ler as notas porque o acesso poderá ser direto.

A solução é colocar métodos que retornem os valores das notas e depois podemos utilizar esses métodos na classe `AlunoFaculdade`.

Observe que tivemos que colocar um método `setAprovado` como privado na classe `aluno`. Esse método é necessário para atribuir uma estado (aprovado ou reprovado) para o atributo `aluno` já que ele é privado.

Foi usado o modificador `private` para que o atributo privado possa ser acessado da classe `AlunoFaculdade`.

Você pode perguntar "por que não usamos o modificador `private`?". Digo que não usamos o `private` porque o modificador `private` disponibiliza acesso somente dentro da MESMA CLASSE."

6. Construa uma aplicação para executar as classes acima.

```
package exercicio3_3;

import javax.swing.JOptionPane;

import com.sun.org.apache.bcel.internal.generic.AALOAD;

public class AppAluno {
    public static void main(String[] args){
        //instanciando um aluno
        Aluno a = new Aluno();

        //configurando o nome e o email do aluno
        a.setNome("Gustavo");
        a.setEMail("gustavo@gmail.com");

        //mostrando o nome e o e-mail do aluno
        JOptionPane.showMessageDialog(null, "Nome: "+a.getNome()+"
Email: "+a.getEMail());

        //Observe que os atributos nome e e-mail são da classe
        Pessoa e foram utilizados métodos da classe Pessoa para
        acessar
        //esses atributos. Os métodos podem ser usados graças a
        herança.

        //Vamos configurar uma série para o aluno
        a.setSerie(3);

        //Vamos configurar as notas do aluno
        a.setNota1(9.6f); //veja que devemos colocar um f (tanto
        faz maiúsculo ou minúsculo) no valor para confirmar que o
        //valor que está sendo inserido é um
        float e não um double. Se for double colocamos d.
        a.setNota2(9.9f);
        a.setNota3(9.9f);
        a.setNota4(10f);

        //vamos mostrar as notas do aluno
        JOptionPane.showMessageDialog(null,"Nota1: "+a.getNota1()
+"\\n"+
        "Nota2: "+a.getNota2()
+"\\n"+
        "Nota3: "+a.getNota3()
+"\\n"+
        "Nota4:
"+a.getNota4());
```

```
//não é necessário mudar de linha. A mudança é apenas uma
questão estética do código
//\n força a mudança de linha

//vamos chamar o método que configura se o aluno foi
aprovado ou não dependendo das notas obtidas
a.setAprovado();

//vamos ver se o aluno foi aprovado ou não
JOptionPane.showMessageDialog(null, "Aprovado:
"+a.getAprovado());

//vamos instanciar um aluno de faculdade.
AlunoFaculdade af = new AlunoFaculdade();

//vamos atribuir um nome ao aluno da faculdade
af.setNome("Pedro"); //veja que através da herança usamos
o método setNome da classe Pessoa. Bacana isso!!

//vamos atribuir as notas para o aluno da faculdade
af.setNota1(5.8f);
af.setNota2(3.90f);
af.setNota3(6.9f);
af.setNota4(3.4f);

//vamos ver se o aluno foi aprovado
af.setAprovado();
//vamos mostrar o nome do aluno da faculdade
JOptionPane.showMessageDialog(null, "Aluno da faculdade:
"+af.getNome());
//vamos mostrar o status do atributo aprovado.
JOptionPane.showMessageDialog(null, "Aprovado:
"+af.getAprovado());
}
}

"faça um teste colocando todas as notas igual a 7 tanto para o
aluno quanto para o aluno da faculdade. Veja que aluno será
aprovado e o aluno da faculdade será reprovado."
```

Exercícios complementares

1. Escreva uma classe chamada Pedido. A classe Pedido deverá possuir os atributos privados: cliente, numeroDoPedido, quantidadeDeItens. Crie um construtor

que utilize como parâmetro o nome do cliente. Um outro construtor deverá possuir o nome do cliente e o número do pedido e um outro construtor deverá possuir como argumento o nome do cliente a quantidade de itens e o número do pedido. Insira os métodos para mostrar cada atributo da classe Pedido

```
package exercicio3_3;

public class Pedido {
    //atributos
    private String cliente;
    private int numeroDoPedido, quantidadeDeItens;

    //construtor 1
    Pedido(String nome){
        this.cliente = nome;
    }

    //construtor 2
    Pedido(String nome, int numero){
        this.cliente = nome;
        this.numeroDoPedido = numero;
    }

    //construtor 3
    Pedido(String nome, int numero, int quantidade){
        this.cliente = nome;
        this.numeroDoPedido = numero;
        this.quantidadeDeItens = quantidade;
    }

    //métodos que retornam conteúdo dos atributos
    public String getCliente(){
        return this.cliente;
    }

    public int getNumeroPedido(){
        return this.numeroDoPedido;
    }

    public int getQuantidadeItens(){
        return this.quantidadeDeItens;
    }
}
```

2. Escreva uma aplicação que instancie 3 objetos sendo que cada um utilize um construtor criado. Mostre todos os atributos para cada objeto instanciado.

package exercicio3_3;

```
import javax.swing.JOptionPane;

public class AppCliente {
    public static void main(String[] args){
        Pedido adina = new Pedido("Gustavo");
        Pedido gvt = new Pedido("Clara", 893);
        Pedido straus = new Pedido("Pedro", 784, 100);

        //mostrando os valores para o objeto Pedido referenciado
        por adina
        JOptionPane.showMessageDialog(null, "Cliente:
"+adina.getCliente());
        JOptionPane.showMessageDialog(null, "Número do pedido:
"+adina.getNumeroPedido());
        JOptionPane.showMessageDialog(null, "Quantidade de
itens: "+adina.getQuantidadeItens());

        //mostrando os valores para o objeto Pedido referenciado
        por gvt
        JOptionPane.showMessageDialog(null, "Cliente:
"+gvt.getCliente());
        JOptionPane.showMessageDialog(null, "Número do pedido:
"+gvt.getNumeroPedido());
        JOptionPane.showMessageDialog(null, "Quantidade de
itens: "+gvt.getQuantidadeItens());

        //mostrando os valores para o objeto Pedido referenciado
        por adina
        JOptionPane.showMessageDialog(null, "Cliente:
"+straus.getCliente());
        JOptionPane.showMessageDialog(null, "Número do pedido:
"+straus.getNumeroPedido());
        JOptionPane.showMessageDialog(null, "Quantidade de
itens: "+straus.getQuantidadeItens());
    }
}
```

3. Crie uma classe Conta, que possua um saldo, e os métodos para configurar saldo, depositar, e retirar. Siga o esqueleto abaixo:

```
class Conta {  
  
    private double saldo;  
  
    void deposita(double x){  
        //..  
    }  
  
    void retira(double x) {  
        //..  
    }  
    double getSaldo() {  
        //..  
    }  
}
```

4. Preencha o esqueleto conforme exemplos vistos no decorrer dos módulos. Crie o método `retira()` que retira um valor da conta e retorne um boolean indicando o sucesso da operação.

```
package exercicio3_3;  
  
public class Conta {  
  
    private double saldo;  
  
    void deposita (double x){  
        if (x > 0){  
            this.saldo = this.saldo + x;  
        }  
    }  
  
    //verifica se o valor a ser retirado não é negativo e se  
    //existe saldo suficiente para ser retirado.  
    boolean retira (double x) {  
        boolean retorno = false;  
        if (x > 0 && this.saldo >= x){  
            this.saldo = this.saldo - x;  
            retorno = true;  
        }  
        return retorno;  
    }  
  
    double getSaldo() {  
        return this.saldo;  
    }  
}
```

```
}
```

5. Adicione um método na classe Conta, que atualiza essa conta de acordo com a taxa selic fornecida.

```
void atualiza(double taxaSelic) {
    this.saldo = this.saldo * (1 + taxaSelic);
}

//início do programa
package exercicio3_3;

public class Conta {

    private double saldo;

    void deposita (double x){
        if (x > 0){
            this.saldo = this.saldo + x;
        }
    }

    //verifica se o valor a ser retirado não é negativo e se
    existe saldo suficiente para ser retirado.
    boolean retira (double x) {
        boolean retorno = false;
        if (x > 0 && this.saldo >= x){
            this.saldo = this.saldo - x;
            retorno = true;
        }
        return retorno;
    }

    double getSaldo() {
        return this.saldo;
    }

    void atualiza(double taxaSelic) {
        this.saldo = this.saldo * (1 + taxaSelic);
    }
}
```

6. Crie duas subclasses da classe Conta: ContaCorrente e ContaPoupanca. Ambas terão o método atualiza reescrito: A ContaCorrente deve atualizar seu saldo com o dobro da taxaSelic, porém subtraindo 15 reais de taxa bancária. A ContaPoupanca

deve atualizar-se com 75% da taxaSelic. Além disso, a ContaCorrente deve reescrever o método deposita, afim de retirar a CPMF de 0.038% a cada depósito.

```
class ContaCorrente extends Conta {
    void atualiza(double taxaSelic){
        // fazer conforme enunciado
    }
    void deposita(double valor) {
        // o deposito deve depositar apenas 99.62% do valor
    }
}

class ContaPoupanca extends Conta {
    void atualiza(double taxaSelic){
        // fazer conforme enunciado
    }
}

//resolução do exercício
package exercicio3_3;

public class ContaCorrente extends Conta{
    //reescrevendo o método deposita
    void deposita (double x){
        if (x > 0){
            double saldo = getSaldo() + x -(x * 0.38);
            super.deposita(saldo);
        }
    }
    //reescrevendo o método atualiza
    void atualiza(double taxaSelic) {
        double saldo = getSaldo() * (2 * taxaSelic) - 15;
        deposita(saldo);
    }
}
```

7.Crie uma aplicação que instacie essas classes, atualize-as e veja o resultado.

```
package exercicio3_3;

import javax.swing.JOptionPane;

public class AppConta {
    public static void main(String[] args){
        //instanciando o objeto conta
        Conta c = new Conta();
    }
}
```

```
//instanciando o objeto ContaCorrente
ContaCorrente cc = new ContaCorrente();

//instanciando o objeto ContaPoupanca
ContaPoupanca cp = new ContaPoupanca();

//utilizando os métodos de Conta
//verificando o saldo
JOptionPane.showMessageDialog(null, "Saldo de conta:
"+c.getSaldo());

//fazendo um depósito
c.deposita(200);

//conferindo o saldo
JOptionPane.showMessageDialog(null, "Saldo de conta:
"+c.getSaldo());

//fazendo uma retirada de conta
c.retira(100);

//conferindo o saldo
JOptionPane.showMessageDialog(null, "Saldo de conta:
"+c.getSaldo());

//utilizando o método atualiza
c.atualiza(12.75);

//conferindo o saldo
JOptionPane.showMessageDialog(null, "Saldo de conta:
"+c.getSaldo());

//utilizando os métodos de ContaCorrente
//depositar
cc.deposita(100);

//conferindo o saldo
JOptionPane.showMessageDialog(null, "Saldo de Conta
corrente: "+cc.getSaldo());

//utilizando o método atualiza
cc.atualiza(12.75);

//conferindo o saldo
JOptionPane.showMessageDialog(null, "Saldo de conta
corrente: "+c.getSaldo());
```

```
//utilizando os métodos de ContaPoupanca
cp.deposita(100);

//conferindo o saldo
JOptionPane.showMessageDialog(null, "Saldo de conta
poupança: "+c.getSaldo());

//utlizando o método atualiza
cp.atualiza(12.75);

//conferindo o saldo
JOptionPane.showMessageDialog(null, "Saldo de conta
poupança: "+c.getSaldo());
}
}
```

Exercícios 3.9

1. Defina classe abstrata.

Classe abstrata é uma classe que possui métodos abstratos isto é métodos que não possuem corpo e deverão ser escritos nas subclasses das classes abstratas.

2. Qual a diferença entre função e funcionalidade. Dê um exemplo.

Função é O QUE um método deve fazer e funcionalidade é COMO um método deve fazer. Ex. função: imprimirRelatório. funcionalidade: algoritmo que estará no método imprimir relatório.

3. O que é reescrita de métodos? Dê um exemplo.

Reescrita de métodos é quando escrevemos um método em uma outra classe com o mesmo nome, mesma assinatura com um corpo (algoritmo) diferente.

Exemplo:

```
String escreve(String nome) {
    return "nome";
}

reescrevendo o método
String escreve(String nome) {
    return "O nome dele é "+nome;
}
```

note que o corpo do método é diferente mas o nome, tipo de retorno e assinatura são iguais.

4. Qual a sintaxe para se declarar uma classe abstrata.

```
abstract class NomeDaClasse{ }
```

5. O que acontece quando uma classe abstrata é instanciada?

Gera um erro de compilação. Mesmo que você conseguisse instanciar uma classe abstrata quando você chamasse o método abstrato o que ele iria fazer?

6. Uma classe que possui um método abstrato tem que ser declarada como abstrata? Justifique sua resposta.

Sim. Porque se ela não for abstrata e for instanciada (supondo que o compilador permitisse), quando você chamar o método abstrato o que ele iria fazer?

7. Como podemos instanciar objetos de uma classe que possui métodos abstratos?

Criando uma classe que herda da classe abstrata e reescrevendo os métodos abstratos. Isto é, escrevendo uma funcionalidade para os métodos abstratos na subclasse.

No computador

1. Implemente uma classe abstrata de nome Forma onde são declarados dois métodos abstratos: float calcularArea(); float calcularPerimetro();

```
package exercicio3_9;

public abstract class Forma {

    public abstract float calculaArea();

    public abstract float calculaPerimetro();

}
```

2. Crie, como subclasse de Forma, uma classe de nome Retangulo cujas instâncias são caracterizadas pelos atributos lado e altura ambos do tipo float. Implemente na classe Retangulo os métodos herdados de Forma e outros que ache necessários.


```
package exercicio3_9;

public class Retangulo extends Forma {
    private float lado, altura;

    public float calculaArea(){
        //funcionalidade
        return lado * altura;
    }

    public float calculaPerimetro(){
        //funcionalidade
        return lado * 2 + altura * 2;
    }

    //outros métodos necessários
    public void setLado(float lado){
        if(lado > 0){
            this.lado = lado;
        }
    }

    public void setAltura(float altura){
        if(altura > 0){
            this.altura = altura;
        }
    }

    public float getLado(){
        return this.lado;
    }

    public float getAltura(){
        return this.altura;
    }
}
```

3. Crie, como subclasse de Forma, uma classe de nome Circulo cujas instâncias são caracterizadas pelo atributo raio do tipo float. Implemente na classe Circulo os métodos herdados de Forma e outros que ache necessários.

```
package exercicio3_9;

public class Circulo extends Forma {
    private float raio;

    public float calculaArea(){
```

```
        return 3.14f * raio * raio;
    }

    public float calculaPerimetro(){
        return 2 * 3.14f * raio;
    }

    //métodos necessários
    public void setRaio(float r){
        if(r > 0 )
            this.raio = r;
    }

    public float getRaio(){
        return this.raio;
    }
}
```

"Observação.

Colocamos f depois do valor de PI para dizer que estamos trabalhando com tipo float.

Não foi necessário usar { depois do if do método setRaio, pois existe somente um comando depois da instrução condicional if."

4. Crie, como subclasse de Retangulo, uma classe de nome Quadrado cujas instâncias são caracterizadas por terem os atributos lado e altura com o mesmo valor.

```
package exercicio3_9;

public class Quadrado extends Retangulo {
    //..cujas instâncias são caracterizadas por terem o lado e
    altura iguais - construtor
    Quadrado(float lado){
        super.setLado(lado);
        super.setAltura(lado);
    }
}
```

5. Elabore um programa de teste onde é declarado um array, de dimensão 5, do tipo estático Forma. Nesse array devem ser guardadas instâncias de Retangulo, Circulo e Quadrado seguindo uma ordem aleatória. Nota: para gerar números aleatórios crie primeiro uma instância da classe Random (presente na biblioteca java.util) e para extrair um inteiro entre 0 e n efetue a invocação nextInt(n). Depois implemente um

ciclo que percorra o array invocando, relativamente a cada um dos objetos guardados, os métodos `calcularArea` e `calcularPerimetro`.

```
package exercicio3_9;

import java.util.*;
import javax.swing.*;

public class AppForma {
    //os objetos instanciados são static porque serão usados
    dentro do método static.
    //instancia objetos como variável de referência a classe
    Objetos. Todas as classes que instanciamos em Java herda da
    //classe Object. Vamos armazenar os objetos Circulo,
    Retangulo e Quadrado que são objetos na classe Object.
    public static Object[] objetos = new Object[3];
    //instancia a classe vetor que também irá armazenar objetos
    por isso ela é do tipo Object.
    public static Object[] vetor = new Object[5];
    //instancia a classe Random do pacote java.util que irá
    gerar números aleatórios.
    public static Random rand = new Random();

    public static void main(String[] args){
        //instancia o objeto da classe Circulo
        Circulo c = new Circulo();
        //instancia o objeto da classe Retangulo
        Retangulo r = new Retangulo();
        //instancia o objeto da classe Quadrado
        Quadrado q = new Quadrado(10);

        //envia valores para os objetos
        c.setRaio(100);
        q.setLado(100);
        r.setAltura(100);
        r.setLado(100);

        //armazena a área dos objetos no vetor objetos que é do
        tipo Object
        objetos[0] = r.calculaArea();
        objetos[1] = c.calculaArea();
        objetos[2] = q.calculaArea();

        //looping para carregar o vetor e calcular a área
        for(int i = 0; i < 5; i++){
            //sorteia um número
```

```
Integer posicao = new Integer(rand.nextInt(3));
//carrega o vetor de objetos com objetos do vetor
objetos. (ambos são object)
vetor[i] = objetos[posicao];
//mostra os objetos carregados (valor de área)
JOptionPane.showMessageDialog(null, "Área
Vetor["+i+"]= "+vetor[i]);
}

//coloca as variáveis de referência aos objetos no vetor
de objetos invocando o método de cálculo de área
objetos[0] = r.calculaPerimetro();
objetos[1] = c.calculaPerimetro();
objetos[2] = q.calculaPerimetro();

//looping para carregar o vetor e calcular o perímetro
for(int i = 0; i < 5; i++){
    //sorteia um número
    Integer posicao = new Integer(rand.nextInt(3));
    vetor[i] = objetos[posicao];
    JOptionPane.showMessageDialog(null, "Perímetro
Vetor["+i+"]= "+vetor[i]);
}
}
}
```

Exercícios da Unidade IV

1. Pela convenção, qual o prefixo utilizado em métodos que retornam boolean?

O prefixo utilizado é o is

2. Quais as principais características de um construtor?

Não possui tipo de retorno e tem o mesmo nome da classe. O construtor pode ou não ter parâmetros.

3. Qual a sintaxe para acessar um atributo estático dentro de um método estático

Utilizamos o nome da classe ponto e o nome do atributo.

4. Escreva um exemplo que utilize um método estático

```
package4_2;

public class Exercicio4{
```

```
static int x;

public static void main(String[] args){
    Exercicio4.x = x + 1;
}
}
```

5. Qual a função do modificador public

Torna atributo, métodos ou classes acessíveis para outras classes.

6. Escreva um exemplo (classe) que utiliza o modificador private e public

```
package4_2;

public class Exercicio6{
    private int k;

    public void setK(int x){
        k = x;
    }
}
```

"O exemplo acima mostra um atributo privado que precisa de um método, nesse caso o setK, para atribuir um valor a k. O método é público e pode ser utilizado por outra classe."

7. Qual a função de um método que possui get como prefixo no nome?

Retornar um valor qualquer.

8. Escreva um programa que utilize um método get.

```
package4_2;

public class Exercicio6{
    private int k;

    public void setK(int x){
        k = x;
    }

    public int getK(){
        return k;
    }
}
```

"Observe que o tipo de retorno do método getK é do mesmo tipo que o atributo k. Dentro do método getK existe também a palavra reservada return. Toda vez que você tiver um tipo de retorno em um método a palavra reservada return estará presente no método."

9. Qual a função do modificador private?

O modificador private torna o atributo, método ou classe (classe interna) acessível somente por um método dentro da própria classe.

10. Escreva um exemplo que utilize o modificador private.

```
package4_2;

public class Exercicio10{
    private String cpf;
}
```

"CPF é um atributo que utilizamos como private porque quando o programador for configurar um CPF ele necessariamente precisará acessar um método que fará a validação do valor inserido e se for realmente um CPF então o valor é atribuído ao atributo CPF."

11. O que é um atributo static?

Quando declaramos um atributo como static dizemos que esse atributo será "visto" por todas as instâncias dessa classe. Isto é, se temos uma classe chamada ClasseA que possua o atributo x como static e instanciarmos 20 objetos DESSA CLASSE, os 20 objetos irão ver o mesmo valor para x. Se a primeira instância (o primeiro objeto) atribuir um valor para x então a 18ª instância (o décimo oitavo objeto) irá ver o mesmo valor de x.

12. O que acontece, em java, quando criamos uma classe e não definimos um construtor para ela?

A linguagem Java já cria um construtor automaticamente. Esse construtor é chamado de construtor padrão. Ele não possui argumentos e seu corpo é vazio.

13. Qual a função de um método que possui set como prefixo no nome?

É um método que irá atribuir um valor a um atributo ou a um outro método.

14. Quais são os três tipos de modificadores de acesso?

public, private, protected.

15. O que é um "acesso direto"?

É quando acessamos diretamente um atributo ou método utilizando o nome da variável de referência um separador ponto e o nome do atributo ou método que queremos utilizar.

16. O que é um construtor?

É um método responsável pela criação de instâncias da classe (objetos).

17. O que acontece com o construtor padrão quando criamos um construtor na classe?

Perdemos o construtor padrão. O primeiro construtor que você cria passa a ser o construtor padrão. Se você quiser o construtor padrão deverá escrevê-lo em seu programa.

18. Atributos são inicializados automaticamente. Qual o valor inicial para tipos numéricos primitivos, String, objetos e boolean?

Tipos numéricos são inicializados com zero;

Tipos String são inicializados com null (Strings são objetos);

Objetos são inicializados com null (não apontam para nenhum objeto na memória);

Tipos boolean são inicializados com false.

No computador

1. Escreva uma classe chamada Pessoa que possua nome e e-mail como atributos privados. Crie os métodos para configurar e mostrar os valores dos atributos.

```
package exercicio4_2;

public class Pessoa {
    private String nome, eMail;

    public void setNome(String n){
        this.nome = n;
    }

    public void setEmail(String em){
        this.eMail = em;
    }
}
```

```
public String getNome() {  
    return this.nome;  
}  
  
public String getEmail() {  
    return this.email;  
}  
}
```

2. Escreva uma classe que se chama Aluno que herda de Pessoa. Adicione ao aluno um atributo privado chamado série, um atributo privado do tipo boolean chamado aprovado, os atributos privados chamados nota1, nota2, nota3 e nota4. O método para configurar a série deverá aceitar somente os valores de 1 a 8. Caso o valor não esteja nessa faixa, zero deverá ser atribuído ao atributo série. O método para configurar os atributos nota1, nota2, nota3 e nota4 deverão aceitar somente números positivos de 1 a 10 caso o número atribuído não esteja nessa faixa, zero deverá ser atribuído ao atributo. O método para atribuir o valor ao atributo aprovado dependerá das notas atribuídas aos atributos notas utilizando o cálculo: $(n1+n2+n3+n4)/4 \geq 6$. Caso a média seja maior ou igual a 7 o atributo receberá verdadeiro (true) caso contrário receberá falso (false).

```
package exercicio4_2;  
  
public class Aluno extends Pessoa {  
    private int serie;  
    private boolean aprovado;  
    private float nota1, nota2, nota3, nota4;  
  
    //método para configurar a série  
    public void setSerie(int n) {  
        if (n >= 1 && n <= 8) {  
            this.serie = n;  
        }  
        else {  
            this.serie = 0;  
        }  
    }  
  
    //método para configurar a nota1  
    public void setNota1(float nota) {  
        if (nota >= 1 && nota <= 10) {  
            this.nota1 = nota;  
        } else {  
            this.nota1 = 0;  
        }  
    }  
}
```



```
    }  
}  
  
//método para configurar a nota2  
public void setNota2(float nota){  
    if (nota >= 1 && nota <= 10){  
        this.nota2 = nota;  
    }else{  
        this.nota2 = 0;  
    }  
}  
  
//método para configurar a nota3  
public void setNota3(float nota){  
    if (nota >= 1 && nota <= 10){  
        this.nota3 = nota;  
    }else{  
        this.nota3 = 0;  
    }  
}  
  
//método para configurar a nota4  
public void setNota4(float nota){  
    if (nota >= 1 && nota <= 10){  
        this.nota4 = nota;  
    }else{  
        this.nota4 = 0;  
    }  
}  
  
//método para configurar aprovação  
protected void setAprovado(boolean estado){  
    this.aprovado = estado;  
}  
  
public void setAprovado(){  
    if((this.nota1 + this.nota2 + this.nota3 + this.nota4)/4  
    >= 6)  
        this.aprovado = true;  
    else  
        this.aprovado = false;  
}  
  
//métodos que retornam valores  
public float getNota1(){  
    return this.nota1;  
}
```

```
public float getNota2(){
    return this.nota2;
}

public float getNota3(){
    return this.nota3;
}

public float getNota4(){
    return this.nota4;
}
//método que retorna o status de aprovação do aluno
public boolean getAprovado(){
    return this.aprovado;
}
}
```

3. Crie uma classe chamada **AlunoFaculdade** que herda de **Aluno** e reescreva o método **aprovado** de forma que a média mude de 6 para 8. Reescreva o método **série** para que o aluno se enquadre na faixa de 1 a 4.

```
//Para resolver esse exercícios teremos que ter na classe
Aluno métodos que retornem os valores das notas.
//Se as notas são privadas em Aluno elas só podem ser
acessadas via métodos da própria classe Aluno.
//Caso você coloque as notas como public então não justifica
escrever métodos para atribuir ou ler as notas porque
//o acesso poderá ser direto.
//A solução é colocar métodos que retornem os valores das
notas e depois podemos utilizar esses métodos
//na classe AlunoFaculdade.
//Observe que tivemos que colocar um método setAprovado como
privado na classe aluno. Esse método é necessário para
//atribuir uma estado (aprovado ou reprovado) para o atributo
aluno já que ele é privado.
//Foi usado o modificador private para que o atributo privado
possa ser acessado da classe AlunoFaculdade.
//Você pode perguntar "por que não usamos o modificador
private?". Digo que não usamos o private porque o modificador
//private disponibiliza acesso somente dentro da MESMA CLASSE.

package exercicio4_2;

public class AlunoFaculdade extends Aluno {
    public void setAprovado(){
```

```
        if((getNota1() + getNota2() + getNota3()+ getNota4())/4 >=
8)
            setAprovado(true);
        else
            setAprovado(false);
        }
    }
```

4. Construa uma aplicação para executar as classes acima.

```
package exercicio4_2;

import javax.swing.JOptionPane;

import com.sun.org.apache.bcel.internal.generic.AALOAD;

public class AppAluno {
    public static void main(String[] args){
        //instanciando um aluno
        Aluno a = new Aluno();

        //configurando o nome e o email do aluno
        a.setNome("Gustavo");
        a.setEmail("gustavo@gmail.com");

        //mostrando o nome e o e-mail do aluno
        JOptionPane.showMessageDialog(null, "Nome: "+a.getNome()+"
Email: "+a.getEmail());

        //Observe que os atributos nome e e-mail são da classe
        Pessoa e foram utilizados métodos da classe Pessoa para
        acessar
        //esses atributos. Os métodos podem ser usados graças a
        herança.

        //Vamos configurar uma série para o aluno
        a.setSerie(3);

        //Vamos configurar as notas do aluno
        a.setNota1(9.6f); //veja que devemos colocar um f (tanto
        faz maiúsculo ou minúsculo) no valor para confirmar que o
        //valor que está sendo inserido é um
        float e não um double. Se for double colocamos d.
        a.setNota2(9.9f);
        a.setNota3(9.9f);
        a.setNota4(10f);
```

```
//vamos mostrar as notas do aluno
JOptionPane.showMessageDialog(null, "Nota1: "+a.getNota1()
+"\n"+
                                "Nota2: "+a.getNota2()
+"\n"+
                                "Nota3: "+a.getNota3()
+"\n"+
                                "Nota4:
"+a.getNota4());
//não é necessário mudar de linha. A mudança é apenas uma
questão estética do código
//\n força a mudança de linha

//vamos chamar o método que configura se o aluno foi
aprovado ou não dependendo das notas obtidas
a.setAprovado();

//vamos ver se o aluno foi aprovado ou não
JOptionPane.showMessageDialog(null, "Aprovado:
"+a.getAprovado());

//vamos instanciar um aluno de faculdade.
AlunoFaculdade af = new AlunoFaculdade();

//vamos atribuir um nome ao aluno da faculdade
af.setNome("Pedro"); //veja que através da herança usamos
o método setNome da classe Pessoa. Bacana isso!!

//vamos atribuir as notas para o aluno da faculdade
af.setNota1(5.8f);
af.setNota2(3.90f);
af.setNota3(6.9f);
af.setNota4(3.4f);

//vamos ver se o aluno foi aprovado
af.setAprovado();
//vamos mostrar o nome do aluno da faculdade
JOptionPane.showMessageDialog(null, "Aluno da faculdade:
"+af.getNome());
//vamos mostrar o status do atributo aprovado.
JOptionPane.showMessageDialog(null, "Aprovado:
"+af.getAprovado());
}
}
```

```
//faça um teste colocando todas as notas igual a 7 tanto para  
o aluno quanto para o aluno da faculdade. Veja que aluno  
//será aprovado e o aluno da faculdade será reprovado.
```

5. (Estudo dirigido)

Vamos construir uma janela gráfica em Java, rever alguns conceitos e aprender novos conceitos.

Vamos criar uma classe que configure uma janela gráfica para isso precisamos importar o pacote `javax.swing.*` (pacotes foram descritos na primeira aula). Vejamos:

```
//importando os pacotes de java (bibliotecas)  
import javax.swing.*;  
public class Janela extends JFrame{  
    //construtor  
    Janela(){  
        //método que configura o tamanho da janela  
        setSize(300,200);  
        //método que configura a localização da janela  
        setLocation(10,20);  
        //método que configura o título da janla  
        setTitle("Minha primeira janela");  
        //método que mostra a janela  
        setVisible(true);  
    }  
}
```

Podemos identificar nessa classe a biblioteca ou pacote que foi importada (`javax.swing.*`) essa linha nos diz que existem classes, representadas por *, dentro de um pacote chamado `swing` que está dentro de outro pacote chamado `javax`. Entenda pacotes como se fossem pastas (diretórios) que serão usadas para a criação da janela. Podemos também identificar os métodos que são iniciados por `set` que são `setSize`, `setLocation`, `setTitle`, `setVisible`, e todos eles possuem parâmetros. que são a largura e altura da janela - `setSize(300,200)` - posição da janela na tela pelas coordenadas x e y - `setLocation(10,20)` - o título da janela - `setTitle("Minha primeira janela")` - e a visibilidade da janela - `setVisible(true)`.

O código abaixo mostra como instanciar a classe acima e torná-la executável. A classe abaixo pode ser chamada de classe executável ou classe programa. Vejamos:

```
public class MostraJanela {  
    public static void main(String[] args){  
        Janela minhaJanela = new Janela();  
    }  
}
```

A classe acima mostra uma classe chamada MostraJanela e essa classe tem um objeto Janela que será instanciado e referenciado pela variável de referência minhaJanela. Como todo o código de construção da janela está no construtor de Janela, então a janela será construída automaticamente e terá a seguinte aparência:

6. (Estudo dirigido)

Escreva uma classe chamada Cliente que possua os atributos privados nome, endereço e cpf do tipo String, idade do tipo int. Sua classe deverá possuir o método setCPF que tem como argumento o cpf. O corpo do método setCPF deve chamar um método chamado validaCPF que tem como argumento o cpf e depois de validado deve atribuir o valor do cpf ao atributo cpf. A classe também deverá ter um método chamado validaCPF que tem como argumento o cpf. O corpo do método validaCPF deve verificar se os caracteres digitados são somente números. Use o método charAt(posição da string) da classe String para ver o valor de cada posição da string.

```
public class Cliente{
    private String nome, endereco, cpf;
    private int idade;

    String validaCPF(String cpf){
        String dominio = "0123456789";
        int cont = 0;
        for(int i=0; i<11; i++){
            for(int j=0; j<10; j++){
                if(cpf.charAt(i) == dominio.charAt(j)){
                    cont++;
                }
            }
        }
        if(cont == 11){
            this.cpf = cpf;
            return cpf;
        }
        else
        {
            return "Erro!";
        }
    }

    void setCPF(String cpf){
        this.cpf = validaCPF(cpf);
    }

    String getCPF(){
        return this.cpf;
    }
}
```

}

O fragmento de código acima que está em *itálico* pode ser substituído por um método chamado `decode` da classe `Integer`. O método `decode(String x)` tem como parâmetro uma `String` e devolve essa `String` se ela contiver apenas números. Caso contrário ela devolve uma exceção (`Exception`). O código abaixo mostra o código acima utilizando o método `decode`.

```
public class Cliente {
    private String nome, endereco, cpf;
    private int idade;

    String validaCPF(String cpf){
        try{
            float x = Float.parseFloat(cpf);
            return ""+cpf;
        }
        catch(NumberFormatException nfe){
            return "Existe caracteres não numéricos na string";
        }
    }

    void setCPF(String cpf){
        this.cpf = validaCPF(cpf);
    }

    String getCPF(){
        return this.cpf;
    }

    void setNome(String nome){
        this.nome = nome;
    }

    void setEndereco(String end){
        this.endereco = end;
    }

    void setIdade(int i){
        if (i > 0){
            this.idade = i;
        }
    }

    String getNome(){
        return this.nome;
    }
}
```

```
String getEndereco() {
    return this.endereco;
}

int getIdade() {
    return this.idade;
}
}
```

Observação importante: Ao utilizarmos os métodos que já existem na linguagem vemos que o trabalho diminui sensivelmente. Conhecer as classes e seus métodos é fundamental para um bom desenvolvimento em Orientação a Objetos.

7. Crie uma classe chamada Conta que possui os seguintes atributos privados saldo, limite do tipo double e dono do tipo Cliente. Construa os getters e setters para os todos atributos. Crie um programa para testar sua classe.

```
package exercicio4_2;

public class Conta {
    private double saldo, limite;
    public Cliente dono = new Cliente();

    public void setSaldo(double s){
        if(s > 0){
            this.saldo = s;
        }
    }

    public void setLimite(double lim){
        if(lim > 0){
            this.limite = lim;
        }
    }
}

//em um outro arquivo....
package exercicio4_2;

import javax.swing.JOptionPane;

public class AppConta {
    public static void main(String[] args){
        Conta conta = new Conta();

        //configurar um nome para o dono da conta
    }
}
```



```
conta.dono.setNome("Gustavo");
//configurar um endereço para o dono da conta
conta.dono.setEndereco("Rua A, 89");
//configurar a idade do dono da conta
conta.dono.setIdade(2);
//configurar o cpf para o dono da conta
conta.dono.setCPF("96748568733");

//mostrando os dados
JOptionPane.showMessageDialog(null, "Nome:
"+conta.dono.getNome());
JOptionPane.showMessageDialog(null, "Endereço:
"+conta.dono.getEndereco());
JOptionPane.showMessageDialog(null, "Idade:
"+conta.dono.getIdade());
JOptionPane.showMessageDialog(null, "CPF:
"+conta.dono.getCPF());
}
```

8. Na classe Conta adicione um construtor que quando utilizado mostra a mensagem "Construindo uma conta" na tela.

```
package exercicio4_2;

import javax.swing.JOptionPane;

public class Conta {
    private double saldo, limite;
    public Cliente dono = new Cliente();

    Conta(){
        JOptionPane.showMessageDialog(null, "Construindo uma
conta");
    }

    public void setSaldo(double s){
        if(s > 0){
            this.saldo = s;
        }
    }

    public void setLimite(double lim){
        if(lim > 0){
            this.limite = lim;
        }
    }
}
```

}

9. Faça com que seu construtor além de mostrar a mensagem na tela, acumule em um atributo `totalContas` a quantidade de conta abertas.

```
public class Conta{
    private double saldo, limite;
    private Cliente dono;
    static int totalconta;

    public Conta(){
        System.out.println("Construindo uma conta");
        totalconta++;
    }

    public void setSaldo(double saldo){
        this.saldo = saldo;
    }

    public double getSaldo(){
        return this.saldo;
    }

    public void setLimite(double limite){
        this.limite = limite;
    }

    public double getLimite(){
        return this.limite;
    }
}

public class AppCliente{
    public static void main(String[] args){
        Cliente c = new Cliente();
        c.setCPF("92695787934");
        System.out.println(c.getCPF());
        Conta conta1 = new Conta();
        Conta conta2 = new Conta();
        conta1.setSaldo(8990);
        System.out.println(conta1.getSaldo());
        System.out.println(conta1.totalconta);

    }
}
```

Exercícios complementares

1. Crie a classe `Empregado` com os atributos `private salario (float)`, `public nome (string)`.

```
package exercicio4_2;

public class Empregado {
    public String nome;
    private float salario;
}
```

2. Crie uma classe chamada `MeuEmpregado`, instancie um funcionário qualquer e tente modificar ou ler o salário. O que acontece?

```
package exercicio4_2;

import javax.swing.JOptionPane;

public class MeuEmpregado {
    public static void main(String[] args){
        Empregado e = new Empregado();
        e.salario = 10;
        e.nome = "Eduarda";
        JOptionPane.showMessageDialog(null, "Salario:
"+e.salario);
        JOptionPane.showMessageDialog(null, "Nome: "+e.nome);

    }
}
```

"Ao tentar acessar um atributo privado (`salario`) o compilador não deixa compilar e emite a mensagem -Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The field `Empregado.salario` is not visible

at `exercicio4_2.MeuEmpregado.main(MeuEmpregado.java:6)`

Isso acontece porque estamos tentando acessar um atributo que é privado (`private`) daquela classe (`Empregado`) e estamos tentando acessá-lo diretamente da classe `MeuEmpregado`."

3. Crie os getters e setters necessários da sua classe `Empregado`

```
package exercicio4_2;

public class Empregado {
```

```
public String nome;
private float salario;

void setSalario(float s){
    salario = s;
}

float getSalario(){
    return salario;
}
}
```

"Foi criado um set para configurar o atributo salário e um get para ler o atributo salário porque ele está declarado como private. Não é necessário criar set e get para nome porque ele é public e pode ser acessado diretamente sem um método."

4. Modifique a sua classe MeuEmpregado que acessam e modificam atributos de um Empregado para utilizar getters e setters.

```
package exercicio4_2;

import javax.swing.JOptionPane;

public class MeuEmpregado {
    public static void main(String[] args){
        Empregado e = new Empregado();
        e.setSalario(10);
        e.nome = "Eduarda";
        JOptionPane.showMessageDialog(null, "Salario:
"+e.getSalario());
        JOptionPane.showMessageDialog(null, "Nome: "+e.nome);

    }
}
```

5. Adicione um atributo na classe Empregado do tipo int que se chama identificador. Esse identificador deve ter um valor único para cada instância do tipo Empregado. O primeiro Empregado instanciado tem identificador 1, o segundo 2 e assim por diante.

```
package exercicio4_2;

public class Empregado {
    public String nome;
    private float salario;
    static private int identificador;
```

```
Empregado() {  
    identificador++;  
}  
  
void setSalario(float s) {  
    salario = s;  
}  
  
float getSalario() {  
    return salario;  
}  
  
int getIdentificador() {  
    return identificador;  
}  
}
```

"Identificador tem que ser privado para que não possamos alterá-lo diretamente. Como não existe um método na classe que permita que você altere manualmente o valor do atributo identificador ele só será alterado quando o construtor for executado. Ele é static porque cada vez que instanciarmos um objeto de Empregad ele será incrementado de uma unidade (veja o código no construtor. Foi criado o método getIdentificador para podermos mostrar o valor corrente do identificador. Como ele é privado então temos que acessá-lo através de método."

6. Crie um getter para o identificador. Você deve criar um setter? Justifique.

```
int getIdentificador() {  
    return identificador;  
}
```

"Não é necessário criar um setter para o identificador porque ele deverá ser incrementado em uma unidade toda vez que um objeto da classe Empregado for instanciado (criado)."

8. Por que o código não compila?

```
class Teste {  
    int x = 45;  
    public static void main(String[] args) {  
        System.out.println(x);  
    }  
}
```

Como devemos resolver esse problema?

"o código acima não compila porque o atributo `int x` está sendo usado dentro de um método estático (`public static void main(String[] args)`). Isto é, temos um método estático tentando acessar um atributo não estático. Para resolver esse problema devemos tornar o atributo `int x` estático - `static int x` - "

Exercícios 5.2.2

1. Escreva o código que exemplifique uma classe final.

```
public final class ClasseA{  
  
}
```

2. O que acontece com um método quando ele é declarado como final?

O método não pode ser reescrito.

3. Qual a diferença de sintaxe de uma classe abstrata e uma interface?

As classes abstratas podem ter métodos concretos e métodos abstratos. As interfaces têm somente métodos abstratos.

4. O que acontece com o compilador ao "enxergar" a palavra `implements` no código de uma classe?

A palavra `implements` diz que devemos implementar todos os métodos da referida interface.

5. Por que de usar interface?

A interface é utilizada para declarar métodos que podem ser usados e devem ser implementados de formas diferentes em classes diferentes com propósitos semelhantes.

A interface também é utilizada para identificar as similaridades entre classes não relacionadas sem ter um relacionamento de classes.

Além disso tudo, a interface tem a capacidade de simular a herança múltipla (que existe em C++) declarando uma classe que implementa diversas interfaces.

6. O que acontece com uma classe quando ela é declarada como final?

Ela não poderá ter subclasses.

7. Na interface todos os métodos são implicitamente `abstract` e `public` e os atributos são implicitamente `static` e `final`. Explique cada um dos modificadores citados na afirmação acima.

Os métodos podem ser acessados por qualquer classe pois são públicos mas devem ser reescritos porque são abstratos (i.e. não têm corpo).

Os atributos são estáticos isto é são vistos por todas as instâncias (objetos) dessa classe e não podem ser alterados porque são constantes (final).

8. Qual a sintaxe para a declaração de uma interface?

```
modificador interface NomeDaInterface{}
```

No computador

1. Escreva uma classe concreta chamada Conta que possua os atributos privados do tipo double saldo, débito e crédito.

```
package exercicio5_2_2;

public class Conta {
    private double saldo, debito, credito;
}
```

2. Crie um método chamado deposito que possua um parâmetro chamado valorDepositado do tipo double em que um usuário poderá fazer um depósito de uma determinada quantia. Um outro método chamado retirada deverá ser criado tendo como parâmetro um valorRetirado. Os dois métodos deverão afetar o atributo saldo.

```
package exercicio5_2_2;

public class Conta {
    private double saldo, debito, credito;

    void deposito(double valorDepositado){
        if(valorDepositado > 0 ){
            this.saldo = this.saldo + valorDepositado;
        }
    }

    void retirada(double valorRetirado){
        if(valorRetirado > 0 ){
            this.saldo = this.saldo - valorRetirado;
        }
    }
}
```

3. Escreva um método chamado extrato para mostrar o valor do saldo atual.

```
package exercicio5_2_2;

public class Conta {
    private double saldo, debito, credito;

    void deposito(double valorDepositado){
        if(valorDepositado > 0 ){
            this.saldo = this.saldo + valorDepositado;
        }
    }

    void retirada(double valorRetirado){
        if(valorRetirado > 0 ){
            this.saldo = this.saldo - valorRetirado;
        }
    }

    double extrato(){
        return this.saldo;
    }
}
```

4. Escreva uma classe que irá instanciar dois objetos de Conta um chamado c1 e outro chamado c2. Faça com que o cliente1 (c1) deposite 100.45 na conta e que o cliente c2 deposite 400.0 na mesma conta. Depois mostre o extrato da conta. Faça o cliente c3 retirar 20.0 da conta e mostre o extrato novamente.

//Para que o depósito seja feito na mesma conta, devemos ter o saldo como static. Vejamos como ficará a classe Conta

```
package exercicio5_2_2;
```

```
public class Conta {
    private static double saldo;
    private double debito, credito;

    void deposito(double valorDepositado){
        if(valorDepositado > 0 ){
            Conta.saldo = Conta.saldo + valorDepositado;
        }
    }

    void retirada(double valorRetirado){
        if(valorRetirado > 0 ){
            Conta.saldo = Conta.saldo - valorRetirado;
        }
    }
}
```



```
double extrato(){
    return Conta.saldo;
}

//em um outro arquivo
package exercicio5_2_2;

import javax.swing.JOptionPane;

public class AppTransacao {
    public static void main(String[] args){
        Conta c1 = new Conta();
        Conta c2 = new Conta();
        Conta c3 = new Conta();

        //depósito de c1
        c1.deposito(100.45);
        c2.deposito(400);
        //extrato
        JOptionPane.showMessageDialog(null, "Extrato da conta
c1: "+c1.extrato());
        JOptionPane.showMessageDialog(null, "Extrato da conta
c2: "+c2.extrato());
        //retirada por c3
        c3.retirada(20);
        //mostra o extrato
        JOptionPane.showMessageDialog(null, "Extrato da conta
c1: "+c1.extrato());
        JOptionPane.showMessageDialog(null, "Extrato da conta
c2: "+c2.extrato());
        JOptionPane.showMessageDialog(null, "Extrato da conta
c2: "+c3.extrato());
    }
}
```

"observe que foram instanciados 3 objetos da classe Conta. Sendo esses referenciados por c1, c2 e c3. Os três objetos instanciados acessam o mesmo atributo saldo porque ele é estático."

5. Escreva uma interface chamada Banco que possui o método `double relatório()`. O método `relatório` é utilizado pelas classes que o implementam para mostrar dados da conta. Essa interface tem um atributo chamado `double limite` com um valor inicial de 100.00

```
package exercicio5_2_2;

public interface Banco {
    double limite = 100;
    public double relatorio();
}
```

6. Na classe Conta o relatório deverá se implementado da seguinte forma: ter uma variável local do tipo double chamada saldoAtual que receberá o limite mais o saldo e imprimirá na tela o saldoAtual.

```
package exercicio5_2_2;

public class Conta implements Banco {
    private static double saldo;
    private double debito, credito;

    public double relatorio(){
        double saldoAtual = Conta.saldo + limite;
        return saldoAtual;
    }

    void deposito(double valorDepositado){
        if(valorDepositado > 0 ){
            Conta.saldo = Conta.saldo + valorDepositado;
        }
    }

    void retirada(double valorRetirado){
        if(valorRetirado > 0 ){
            Conta.saldo = Conta.saldo - valorRetirado;
        }
    }

    double extrato(){
        return Conta.saldo;
    }
}
```

7. Faça as seguintes modificações no exercício anterior:

Não coloque o saldo como static, compile e execute o programa. Anote a mensagem de erro.

Exception in thread "main" java.lang.Error: Unresolved compilation problems:

```
Cannot make a static reference to the non-static field
Conta.saldo
Cannot make a static reference to the non-static field
Conta.saldo
at exercicio5_2_2.Conta.deposito(Conta.java:14)
at exercicio5_2_2.AppTransacao.main(AppTransacao.java:12)
```

8. Escreva um programa que desenhe uma janela com um botão no centro do monitor. Ao ser pressionado o botão, seu nome deverá ser mostrado na tela em uma janela de diálogo.

```
package exercicio5_2_2;
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class Janela {
    public JFrame janela = new JFrame("Exercício");
    public JButton botao = new JButton("Pressione");
    public JPanel painel = new JPanel();

    public Janela(){
        painel.setLayout(null);
        TrataEvento te = new TrataEvento();
        botao.setBounds(100, 160, 180, 25);
        botao.addActionListener(te);
        painel.add(botao);

        janela.setSize(400,400);
        janela.setLocation(500, 200);
        janela.add(painel);
        janela.setVisible(true);
    }

    //classe interna TrataEvento
    private class TrataEvento implements ActionListener{
        public void actionPerformed(ActionEvent e) {
            if (e.getSource()== botao){
                JOptionPane.showMessageDialog(janela, "Curso de
Programação Orientada a Objetos");
            }
        }
    }

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

```
Janela j = new Janela();  
}  
}
```

5.4 Exercícios

1. Crie uma classe chamada Automóvel que possua um construtor que imprime a string “Eu sou um automóvel”. A classe Automóvel deverá possuir um método chamado combustivel que retorna a String “Uso combustivel” e um método chamado numeroRodas que retorna a String “Tenho rodas”

```
package exercicio5_4;  
  
public class Automovel {  
    Automovel() {  
        System.out.println("Eu sou um automóvel");  
    }  
  
    public String combustivel() {  
        return "uso combustivel";  
    }  
  
    public String numRodas() {  
        return "tenho rodas";  
    }  
}
```

2. Crie uma classe chamada Carro que estende da classe Automóvel que possua um construtor que imprime a string "Eu sou um carro!", um método chamado combustivel que retorna a string "Uso álcool" e um método chamado numeroRodas que retorna a string "Tenho 4 rodas!".

```
package exercicio5_4;  
  
public class Carro extends Automovel {  
  
    Carro() {  
        System.out.println("eu sou um carro!");  
    }  
  
    public String combustivel() {  
        return "uso alcool";  
    }  
}
```

```
        public String numRodas(){
            return "tenho 4 rodas!";
        }
    }
}
```

3. Crie uma classe chamada Moto que estende da classe Automóvel que possua um construtor que imprime a string "Eu sou uma moto!", um método chamado combustivel que retorna a string "Uso gasolina!" e um método chamado numeroRodas que retorna a string "Tenho 2 rodas!".

```
package exercicio5_4;

public class Moto extends Automovel{
    Moto(){
        System.out.println("Sou uma moto!");
    }

    public String combustivel(){
        return "uso gasolina";
    }

    public String numRodas(){
        return "tenho 2 rodas";
    }
}
}
```

4. Crie uma classe chamada AplicacaoPolimorfismo que possui um método abaixo:

```
public static void imprime(Automovel x){
    System.out.println(x.combustivel());
    System.out.println(x.numeroRodas());
}
}
```

Essa classe também possui o método public static void main(String[]args) que instancia os objetos:

```
//instanciando um automovel
Automovel a = new Automovel();
System.out.println("-----");
//instanciando um carro
Automovel c = new Carro();
System.out.println("-----");
//instanciando uma moto
Automovel m = new Moto();
```

```
System.out.println("-----");
Ela deverá também chamar o método imprime da seguinte maneira:
//chama o método imprime()
imprime(a);
System.out.println("-----");
imprime(c);
System.out.println("-----");
imprime(m);
```

Execute o exercício acima e escreva suas conclusões sobre os resultados obtidos.

```
package exercicio5_4;

public class AplicacaoPolimorfismo {

    public static void imprime(Automovel x){
        System.out.println(x.combustivel());
        System.out.println(x.numRodas());
    }

    public static void main(String[] args){
        //instanciando um automovel
        Automovel a = new Automovel();
        System.out.println("-----");
        //instanciando um carro
        Automovel c = new Carro();
        System.out.println("-----");
        //instanciando uma moto
        Automovel m = new Moto();
        System.out.println("-----");

        //chama o método imprime()
        imprime(a);
        System.out.println("-----");
        imprime(c);
        System.out.println("-----");
        imprime(m);
    }
}
```

REFERÊNCIAS

[1] – Deitel, H.M., Java, como programar, 4.ed. – Porto Alegre: Bookman, 2003.

- [2] – Sierra, K., Bates, B., Certificação Sun para programador java 5: Guia de estudo, 4.ed. – Rio de Janeiro: AltaBooks, 2006.
- [3] – Puga, S., Rosseti, G. Lógica de programação e estrutura de dados, com aplicações em Java, São Paulo: Prentice Hall, 2003.
- [4] – Barnes, D.J., Kölling, M., Programação orientada a objetos com java: Uma introdução utilizando o Blue J., São Paulo: Pearson Prentice Hall, 2004.
- [5] – Thompson, M.A., Java 2 & banco de dados: aprenda na prática a usar java e SQL para acessar banco de dados relacionais., São Paulo: +rica, 2002.
- [6] – Keogh, J., Giannini, M., OOP Desmystified, McGraw-Hill/Osborne, 2004.
- [7] – Leopoldino, F. L., Instalando o J2SE 5.0 JDK no Windows 2000/XP, site na internet no endereço:
<http://www.guj.com.br/content/articles/installation/j2sdkinstall.pdf> acessado em 16/06/2007.
- [8] – Wikipédia, site na internet acessado em 16/06/2007
http://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos
- [9] - Caelum: Ensino e Soluções em Java, site na internet no endereço:
www.caelum.com.br, acessado em 19 de Dezembro de 2007.