Programação Orientada a Objetos

UNIDADE 3

GUIA DE ESTUDO

UNIDADE 3

PROGRAMAÇÃO ORIENTADA A OBJETOS

PARA INÍCIO DE CONVERSA

Olá, caro(a) estudante! Vamos dar início à unidade 3 do nosso curso de Programação Orientada a Objetos. Nesse guia de estudo da respectiva disciplina, vamos falar principalmente sobre abstração de dados, encapsulamento e mecanismo de herança. Veremos também a criação e uso de hierarquia de classes, interfaces de classes e classes abstratas. Conto com sua determinação nesta nova jornada de estudo!

ORIENTAÇÕES DA DISCIPLINA

Nesta unidade vamos estudar os seguintes tópicos:

1. Abstração e encapsulamento;
2. Herança;
3. Criação e uso de hierarquia de classes;
4. Classes abstratas e Interfaces.

Vamos começar!

PALAVRAS DO PROFESSOR

Prezado(a) aluno(a), para entender os conceitos de abstração e encapsulamento, vamos ver um mecanismo que já era bastante usado em programação estruturada, a modularização.

Quando criamos um módulo (uma função ou um procedimento) em programação estruturada, estamos usando uma estratégia para resolver o problema diminuindo esse em partes menores. Na hora que criamos um procedimento para fazer uma tarefa qualquer, programamos dentro desse módulo os comandos necessários para a execução de tal tarefa. Damos um nome a esse procedimento e quando precisamos executar esse, colocamos uma chamada em nosso código.

Uma vez implementado esse procedimento, precisamos apenas saber O QUE ele faz para usá-lo. Não precisamos nesse momento saber COMO foi feito a codificação do mesmo. Chamamos isso de abstração procedimental. Como isso, nos concentramos na solução do problema maior sem pensar nos detalhes de como aquele procedimento foi construído.

Seguindo esse mesmo raciocínio, podemos expandir o conceito de abstração para uma forma mais completa. Usaremos em orientação a objeto o conceito de abstração de dados. Em um tipo abstrato de dados (Abstract Data Type – ADTs) temos uma estrutura de dados mais sofisticada, abstrata, que contém a representação de dados e também as operações que podem ser realizadas sobre esses dados. Dessa forma temos um ocultamento de informações, pois a maneira como esses dados são representados e como as operações associadas aos dados não precisam ser visíveis. Basta saber como se comportam e usar de forma correta quando necessário.

Em orientação a objeto, normalmente, as classes ocultam os detalhes de implementação dos seus usuários (outras classes). Com isso, tentamos deixar a classe mais completa e independente possível. Os detalhes são ocultos e protegidos dos usuários, sendo acessados de forma controlada e segura.

Quando criamos um objeto, queremos que seus atributos, características, estejam com valores válidos. Ou seja, é importante que esses dados estejam consistentes e não assuma valores inapropriados. Por exemplo, quando criamos um objeto que representa uma pessoa, e que esse tem um atributo idade, não é interessante que esse atributo possa assumir um valor negativo (-19, por exemplo), pois ninguém possui idade negativa.

Usamos então um mecanismo para proteger esse tipo de situação. Usamos atributos privados e acessamos esses através de métodos assessores (get/set) para garantir a integridade das informações dentro dos objetos. Portanto, temos um encapsulamento das informações.

VOCÊ SABIA?

O encapsulamento separa os aspectos externos de um objeto dos detalhes de implementação. Permite que uma aplicação seja construída a partir de um conjunto de componentes que realizam seu trabalho de forma independente. Cada componente contém tudo que precisa para realizar seu trabalho independentemente dos outros componentes.

Com isso, estimulamos a divisão de responsabilidades: cada objeto faz apenas o que deve fazer, não realizando tarefas extras. A independência entre os objetos é obtida escondendo detalhes internos do funcionamento de cada componente, tornando o uso do componente acessível apenas através de uma interface externa. Com o encapsulamento temos um controle de acesso maior às informações do estado interno do objeto. Para isso, usamos métodos públicos para acessar os atributos privados onde as informações estão armazenadas.

Você não precisa saber como um telefone realmente funciona, para poder usá-lo. Só precisa saber qual a funcionalidade desse aparelho, e conhecer sua interface, ou seja, a forma de nos comunicarmos com ele. Se a companhia telefônica mudar seus processos, ou o fabricante criar modelos novos de aparelhos telefônicos, você vai continuar usando esse aparelho normalmente.

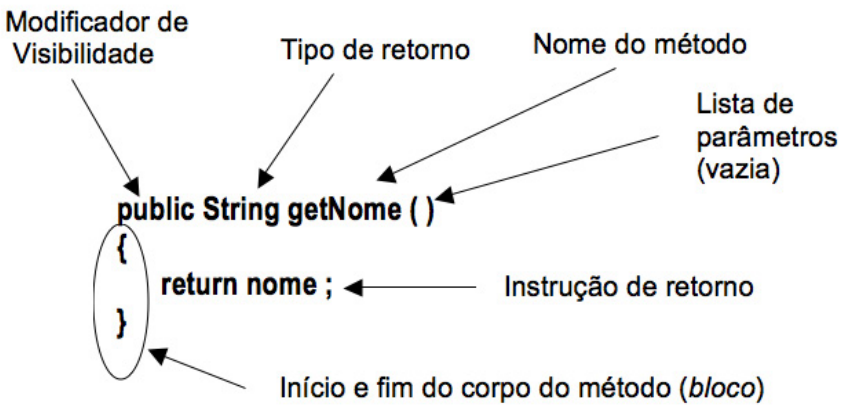
Na época em que os sistemas de transmissão dos celulares migraram de analógico para digital, os usuários continuaram fazendo suas ligações do mesmo jeito. Não foi necessária nenhuma mudança na forma que se usava o celular, seja ele analógico ou digital. Em outras palavras, a implementação foi alterada, mas a interface permaneceu a mesma. De forma analógica, a implementação de uma classe, pode ser alterada sem afetar a sua interface.

A interface será a lista de serviços oferecidos por uma classe de objetos. É formada por métodos e atributos públicos, ou seja, podem ser acessados por outros objetos. Funciona como um "contrato" que o objeto estabelece com o mundo exterior que define exatamente como este pode fazer uso do objeto. Funciona como um "painel de controle" do objeto. Na interface não temos a informação de como os trabalhos são realizados dentro dos objetos. Dessa forma, temos a liberdade para modificar o funcionamento do método sem "quebra de contrato" com o mundo exterior.

Em sistemas orientados a objetos, é praticamente consenso entre os vários autores que, todos os atributos devem ser construídos como privados. Podendo ser acessados ou alterados apenas através de operações na interface pública, ou seja, utilizando métodos públicos. Esses métodos são conhecidos como métodos de acesso e métodos modificadores. Juntos, esses métodos viabilizam o encapsulamento. Vamos conhecer melhor os métodos de acesso e métodos modificadores:

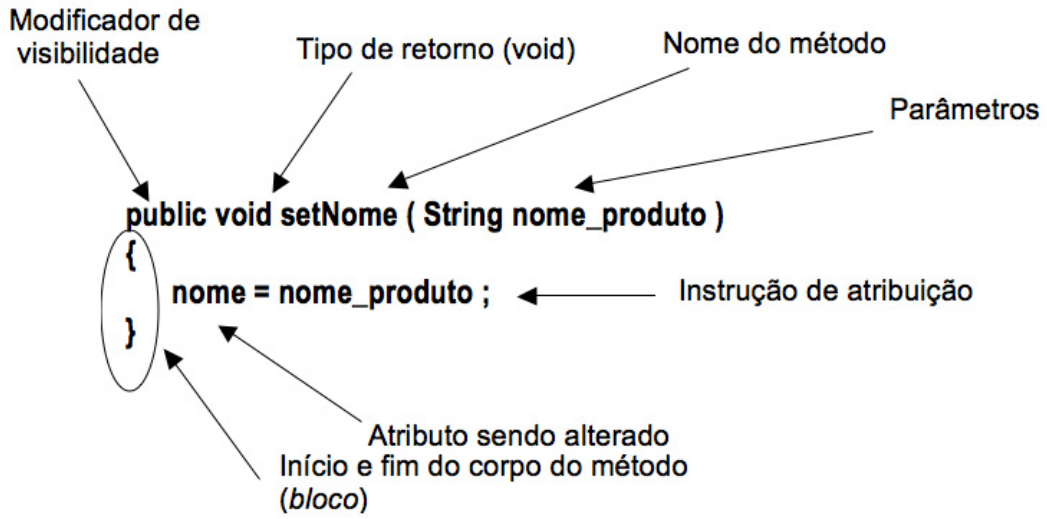
Os métodos de acesso permitem acessar os atributos do objeto que estão declarados como privados. Normalmente, os nomes desses métodos começam com "get" seguido do nome do atributo.

Veja com seria um método de acesso para um atributo "nome":



Os **métodos modificadores** permitem modificar os atributos do objeto que estão declarados como privados. Modifica o estado de um objeto por meio da modificação do valor de um ou mais atributos do objeto. Geralmente, contêm instruções de atribuição e recebem parâmetros. Normalmente, os nomes desses métodos começam com **"set"** seguido do nome do atributo.

Veja com seria um método de acesso para um atributo "nome":



Em JAVA temos os modificadores de visibilidade que restringem o acesso aos atributos, classes e métodos. Temos então os seguintes modificadores:

- Modificador de acesso public - O uso do modificador public indica que o atributo, a classe, método assim declarada possa ser acessada em qualquer lugar e a qualquer momento da execução do programa.
- Modificador de acesso private - O modificador de acesso private quando aplicado a um atributo ou a um método indica que esses só podem ser acessados de dentro da classe onde foi criado (encapsulamento). No caso de uma classe que herde de uma superclasse com atributos declarados como private só poderá ter acesso a eles e através dos métodos públicos de acesso da própria superclasse, caso contrário, não haverá acesso a estes atributos fora da classe.
- Modificador de acesso protected - O modificador protected indica que o método ou atributos assim declarado possa ser acessado somente dentro do pacote em que está contida através de uma subclasse.

Veremos agora algumas vantagens do encapsulamento:

- Reutilização – Uma vez que o componente é independente, ele pode ser utilizado em diferentes programas com facilidade. O nível de reutilização nesse caso é bem mais forte, pois cada objeto tende a ser completo e independente, facilitando a reutilização.
- Divisão de responsabilidades – Cada objeto faz apenas o que lhe é devido, não assumindo outras funções extras.
- Facilidade de modificação – Uma vez que o mundo exterior não conhece a implementação do componente, esta pode ser alterada sem problemas.
- Ausência de efeitos colaterais – Uma vez que o componente não depende de partes do seu programa para funcionar, não há risco de ele modificar o funcionamento do programa.

DICA

Considere o exemplo da classe Pessoa que vimos no guia da unidade 2. Nessa classe temos métodos modificadores que não permitem que valores inválidos sejam atribuídos. Perceba que no construtor, linhas 11 e 13, são feitas chamadas a esses métodos. Com isso, apenas valores válidos serão colocados em idade e altura. Para ilustrar, colocamos um pedaço da classe Pessoa, os construtores e os métodos modificadores.

Nesse exemplo todos os atributos são privados (private), ou seja, só poderão ser acessados através de métodos públicos. Essa é uma implementação com encapsulamento.

EXEMPLO

Pessoa.java

```
1  public class Pessoa {
2
           private String nome;      // Nome completo da
3  pessoa.
4          private int idade;        // Idade da pessoa.
           private char sexo;        // Sexo da pessoa: 'M'
5  para masculino e 'F' para feminino.
           private double altura;    // Altura da pessoa: valor
6  real expresso em metros.
7
           // Construtor da classe Pessoa com quatro parâ-
8  metros.
           public Pessoa ( String n , int i , char s , double a
9  ) {
10                 nome = n ;
11                 setIdade ( i ) ;
12                 sexo = s ;
13                 setAltura( a ) ;
14         } // Fim do construtor.
15
16         // Construtor da classe Pessoa sem parâmetros.
17         // Definido explicitamente para substituir o
18         // construtor vazio padrão do compilador JAVA
19         public Pessoa ( ) {
                   // Vazio, apenas para substituir o constru-
20 tor padrão !
21         }
22
           // Construtor da classe Pessoa com dois parâme-
23 tros, apenas o nome e a idade.
24         // Método de acesso para o campo idade.
25         public void setIdade ( int idade ) {
26                 if ( idade >= 0 ) {
27                         this.idade = idade ;
28                 }
29                 else {
```

```
                        System.out.println ( " Idade in-
30 válida ! " ) ;
31                  }
32          } // fim do método setIdade.
33                  else {
                        System.out.println ( " Idade in-
34 válida ! " ) ;
35                  }
36          } // fim do método setIdade.
37
38          // Método de acesso para o campo altura.
39          public void setAltura ( double altura ) {
40                  if ( altura > 0 ) {
41                      this.altura = altura ;
42                  }
43                  else {
                        System.out.println ( " Altura invá-
44 lida ! " ) ;
45                  }
46          } // fim do método setAltura.
47
48          // Outros métodos . . .
49 } // Fim da classe Pessoa.
```

Fonte: Adaptação do Autor

No caso em que não utilizasse o encapsulamento, os atributos seriam públicos e não usaríamos os métodos modificadores e de acesso. O acesso aos atributos seria feito diretamente com a operação de ponto nome do atributo (Ex.: p.idade = 21 ;). Como não está validando os dados que serão armazenados nos atributos, poderíamos atribuir valores inválidos, por exemplo, p.idade = -25;. Neste exemplo, a pessoa ficaria com uma idade negativa!

Abaixo segue uma versão da classe Pessoa, vista anteriormente, sem encapsulamento.

Pessoa.java

```
1  public class Pessoa {
2
3           public String nome;      // Nome completo da pessoa.
4           public int idade;        // Idade da pessoa.
            public char sexo;        // Sexo da pessoa: 'M' para
5  masculino e 'F' para feminino.
            public double altura;    // Altura da pessoa: valor real
6  expresso em metros.
7
8           // Construtor da classe Pessoa com quatro parâmetros.
9           public Pessoa ( String n , int i , char s , double a ) {
10                  nome = n ;
```

```
11                      Idade = i ;
12                      sexo = s ;
13                      altura = a ;
14              } // Fim do construtor.
15
16              // Construtor da classe Pessoa sem parâmetros.
17              // Definido explicitamente para substituir o
18              // construtor vazio padrão do compilador JAVA.
19              public Pessoa ( ) {
                        // Vazio, apenas para substituir o construtor pa-
20  drão !
21              }
22
                // Construtor da classe Pessoa com dois parâmetros,
23  apenas o nome e a idade.
24              public Pessoa ( String nome , int idade ) {
25                      this.nome = nome ;
26                      this.idade = idade  ;
27              } // Fim do construtor.
28
29              // Outros métodos . . .
30
                // Método que exibe na tela de console as informações
31  da pessoa.
32              public void exibe ( ) {
33                      System.out.println ( " Nome   : " + nome ) ;
34                      System.out.println ( " Idade  : " + idade ) ;
35                      System.out.println ( " Sexo   : " + sexo ) ;
36                      System.out.println ( " Altura : " + altura ) ;
37              } // fim do método exibe.
38
39              // Método que soma mais um ano na idade da pessoa.
40              public void aniversario ( ) {
41                      idade = idade + 1 ;
42              } // fim do método aniversario.
43
44  } // Fim da classe Pessoa.
```

Fonte: Adaptação do Autor

HERANÇA

Veremos agora mais um mecanismo muito importante na programação orientada a objeto, a herança. A herança é o mecanismo para definir uma nova classe em termos de uma já existente.

LEITURA COMPLEMENTAR

Como guia para seus estudos, recomendo que você leia o capítulo de herança do livro do DEITEL (DEITEL. Java Como Programar. Ed. Pearson Brasil, 2005.), capítulo 9.

Boa Leitura!

A herança é um mecanismo muito importante para a reutilização de códigos. Dentre os níveis de reutilização, é um dos mais completos. Temos a reutilização através de cópia simples de código (copiar e colar), onde se modifica a cópia do código para se adaptar as necessidades. Temos também o uso de módulos, bibliotecas de funções, que é um pouco mais sofisticado, pois pode usar parâmetros nas funções para tornar a reutilização mais flexível. Mas sem dúvida, o uso de herança em orientação a objetos é a forma mais completa e flexível de reutilização.

A herança estabelece um relacionamento entre classes de objetos de forma que permita a uma classe possuir atributos e operações definidas por outra classe mais genérica. A classe mãe é chamada de superclasse e a classe filha de subclasse.

Para criar uma classe herdeira de outra em JAVA, usamos após o nome da classe herdeira (subclasse) a palavra reservada extends seguido do nome da superclasse. Suponha que você deseja criar uma classe Cliente herdeira da classe Pessoa. Ou seja, Pessoa é a superclasse e Cliente é a subclasse.

Declaração da subclasse:

```
public class Cliente extends Pessoa {
. . .
}
```

Como a subclasse herda todos os atributos e métodos da superclasse, só é necessário acrescentar os novos métodos particulares dessa classe herdeira e sobrescrever os métodos que precisam ser adaptados para essa subclasse. É importante lembrar que não é possível remover métodos nas classes herdeiras. Caso seja importante a remoção de algum método da superclasse na subclasse, pode ser que essa relação de herança não esteja correta. Nesse caso é bom avaliar novamente o projeto.

Uma subclasse pode possuir os seguintes tipos de métodos:

- Herdados: Métodos que a subclasse adquiriu da superclasse.
- Sobrescritos: Métodos herdados que foram redefinidos na subclasse.
- Novos: Métodos adicionados na subclasse.

FIQUE ATENTO!

Como vimos anteriormente, uma subclasse pode modificar o funcionamento de métodos herdados sobrescrevendo os mesmos (overriding). O método novo (sobrescrito) substitui o código antigo presente na superclasse. Apenas métodos com acesso public e protected podem ser sobrescritos. Se um método da subclasse sobrescrever um método da superclasse, a "visibilidade" do método não pode ser reduzida (um método não

pode ser sobrescrito para se tornar menos acessível). Ou seja:

- Métodos protected podem ser declarados como protected ou public.
- Métodos public só podem ser declarados como public.

EXEMPLO

Esse exemplo é uma adaptação do exemplo de herança do capítulo 9 do livro do Deitel, Java: como programar. Imagine um programa para gerenciar a folha de pagamento de uma empresa. Tal programa pode conter uma classe Empregado.

A classe empregado é definida para o empregado comum, que possui um salário fixo. Entretanto, a empresa também possui empregados que além do salário fixo, uma comissão sobre as vendas realizadas. Essa é a única diferença entre os dois tipos de funcionários.

Como implementar no programa um suporte a esse funcionário comissionado?

Considere o código abaixo que representa a classe Empregado. Essa classe modela um empregado simples que tem um salário fixo, além das informações de nome e CPF. Com essa classe podemos criar objetos que representam empregados que tem remuneração fixa.

Empregado.java

```
1  public class Empregado {
2
           private String nome ;      // Nome completo do
3  empregado.
4          private int cpf ;          // CPF do empregado.
           private double salario ;   // Valor do salário do
5  empregado.
6
           // Construtor da classe Empregado com três parâ-
7  metros.
           public Empregado ( String nome_emp , int cpf ,
8  double val ) {
9            nome = nome_emp ;
10           this.cpf = cpf ;
11           setSalario ( val ) ;
12         } // Fim do construtor.
13
14         // Método de acesso para o campo salário.
15         public double getSalario( ) {
16           return salario;
17         } // fim do método getSalario.
18
19         // Método de acesso para o campo salário.
```

```
20          public void setSalario( double salario ) {
21             if ( salario >= 0 )
22                 this.salario = salario ;
23              else
                    System.out.println ( " Não pode ter salário
24  com valor negativo ! " ) ;
25          } // fim do método setSalario.
26
27          // Método de acesso para o campo CPF.
28          public int getCpf ( ) {
29                  return cpf ;
30          } // fim do método getCpf.
31
32          // Método de acesso para o campo CPF.
33          public void setCpf ( int cpf ) {
                    // Antes de atribuir o CPF deve ser feita
34  uma validação do mesmo !
35                  this.cpf = cpf ;
36          } // fim do método setCpf.
37
38          // Método de acesso para o campo nome.
39          public void setNome ( String nome ) {
40                  this.nome = nome ;
41          } // fim do método setNome.
42
43          // Método de acesso para o campo salário.
44          public String getNome ( )
45    {
46          return nome ;
47    } // fim do método getNome.
48
49          // Outros métodos omitidos para simplificar.
50
51  }
```

Fonte: Adaptação do Autor.

Com base nessa classe vamos implementar no nosso programa o suporte para representar os funcionários que trabalham com remuneração fixa mais as comissões. Para isso, vamos criar uma classe que modela os empregados que recebem comissão. Essa classe seria chamada de EmpregadoComComissao. Para codificar essa nova classe pode ser utilizada uma solução apenas com encapsulamento.

Para a solução utilizando apenas encapsulamento, seria copiado todo o código da classe Empregado para uma nova classe EmpregadoComComissao, modificando a mesma para adicionar o suporte a comissões. O problema desta solução está na necessidade de manter dois códigos: sempre que for necessário adicionar/corrigir a classe Empregado, será necessário corrigir a classe EmpregadoComComissao também.

A melhor solução é utilizar herança. Na solução utilizando herança, seria criada a classe EmpregadoComComissao herdeira de Empregado, obtendo todos os métodos e atributos definidos na superclasse Empregado. Nessa classe herdeira só é necessário codificar os atributos e métodos que são pertinentes aos empregados com comissão. O restante é herdado automaticamente através do mecanismo de herança.

Uma vez que um objeto do tipo EmpregadoComComissao é também um objeto do tipo Empregado, ele pode receber quaisquer mensagens que seriam enviadas a um objeto do tipo Empregado.

Abaixo, temos uma implementação simplificada da classe EmpregadoComComissao, que foi feita para a solução usando herança.

EmpregadoComComissao.java

```
1  // Classe para o Empregado com comissão.
2  // Essa classe estende a classe de Empregado.
3  // É uma subclasse de Empregado.
4
5  public class EmpregadoComComissao extends Empregado {
6
           private double comissao ;        // Valor da comissão por produto
7  vendido.
8          private int numero_vendas ;      // Número de produtos vendidos.
9
           // Construtor da classe EmpregadoComComissao com quatro
10 parâmetros.
       public EmpregadoComComissao ( String n_emp , int c , double val ,
11 double com ) {
            // Chamada ao método construtor da superclasse ( Empregado
12 ).
            super ( n_emp , c , val) ;
13          if  ( com > 0 )
14              comissao = com ;
15      } // Fim do construtor.
16
17
           // Método para adicionar mais uma quantidade de produtos ven-
18 didos.
    public void adicionarVendas ( int vendas ) {
19      numero_vendas = numero_vendas + vendas;
20  } // fim do método adicionarVendas.
21
22
23  // Método para calcular o pagamento do empregado.
24  // leva em consideração o valor do salário fixo.
25  // adicionado do valor de comissão por produto vendido.
26  public double calcularPagamento ( ) {
27          return getSalario ( )  + comissao * numero_vendas ;
28  } // fim do método calcularPagamento.
29
```

```
30          // Método de acesso para o campo comissão.
31      public double getComissao ( ) {
32          return comissao ;
33      } // fim do método getComissao.
34
35          // Método de acesso para o campo comissão.
36      public void setComissao ( double comissao ) {
37          if  ( comissao >= 0 )
38              this.comissao = comissao ;
39          else
40              System.out.println ( " Comissão não pode ser negativa ! " ) ;
41      } // fim do método setComissao.
42
43      // Método de acesso para o campo numero_vendas.
44      public int getNumero_vendas ( ) {
45          return numero_vendas ;
46      } // fim do método getNumero_vendas.
47
48      // Método de acesso para o campo numero_vendas.
49      public void setNumero_vendas ( int numeroVendas ) {
50          if  ( numeroVendas >= 0 )
51                  numero_vendas = numeroVendas ;
52          else
                    System.out.println ( " Número de produtos vendidos não
53  pode ser negativo ! " ) ;
54      } // fim do método setNumero_vendas.
55
56      // Outros métodos omitidos para simplificar.
57
58  }
```

Fonte:Adaptação do Autor 2016.

Porque utilizar herança?

- Nem sempre encapsulamento é a melhor solução.
- Possibilidade de alterar o funcionamento herdado da classe mãe (superclasse), criando uma classe mais adequada a um tipo de situação (subclasse), sem precisar mexer na classe original (superclasse).
- Possibilidade de criar classes mais específicas, reutilizando código de classes genéricas, sem precisar copiar e modificar o modificar, parte do código colado.

GUARDE ESSA IDEIA!

Em JAVA o uso do modificador final pode ser usado também em classes. Como você deve ter visto antes, em outras disciplinas e programação com JAVA, o modificador final é usado para declarar constantes. Ou seja, um identificador declarado de um determinado tipo, quando precedido pelo modificador final, não pode mais ter seu valor alterado. Funcionado como uma constante. Se o modificador final for aplicado em um método, isso faz com que esse método não possa ser redefinido em subclasses derivadas. Em classes, o modificador final impossibilita que essa classe possa ser base para a criação de uma subclasse a partir dela. Por exemplo, se nós inserirmos o modificador final na declaração da classe Empregado que usamos no exemplo anterior, essa não poderia ser usada como superclasse para a criação da subclasse EmpregadoComComissao.

Se:

```
public final class Empregado {  . . .
```

Então a declaração abaixo daria erro:

```
public class EmpregadoComComissao extends Empregado { . . .
```

Teste depois para comprovar!

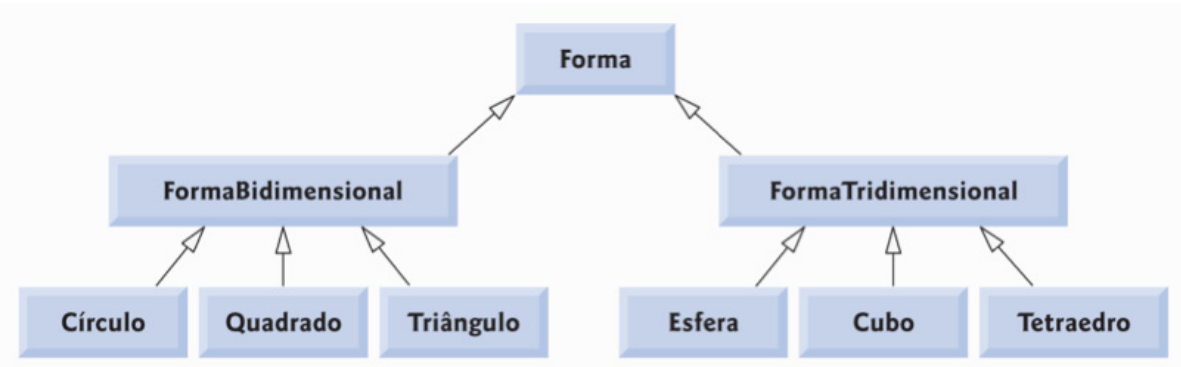
Criação e uso de hierarquia de classes

Uma vez que subclasses podem ser superclasses de novas classes, podemos definir uma hierarquia de classes em nossos programas. Podemos entender a hierarquia de classe como uma árvore que contém todas as classes que estão relacionadas através de herança. A hierarquia de classes em Java se inicia com a classe Object, definida em java.lang. Uma classe herda atributos e métodos de todos os seus ancestrais.

Em Java, uma classe pode herdar diretamente de apenas uma classe (herança simples). E possível, entretanto, implementar várias interfaces.

EXEMPLO

Abaixo, teremos um exemplo que ilustra uma hierarquia de classe:



Hierarquia de herança para Forma.

Fonte: DEITEL. Java Como Programar. Ed. Pearson Brasil, 2005. Capítulo 9. Figura 9.3.

Como podemos saber quando devemos utilizar herança e não encapsulamento? Como saber se a hierarquia de classes está correta?

A forma de validar herança é usar a frase "é um". Exemplo: gato e animal mamífero. Todo gato é um animal mamífero. Todo mamífero é um gato (?). Dessa forma podemos verificar essa relação. Validar a herança implica em verificar se faz sentido tratar a classe filha da mesma forma que se trata a classe mãe.

Observe que uma classe filha pode apenas adicionar funcionalidade em relação à classe mãe. Remover funcionalidade não é permitido. Caso haja necessidade de remover funcionalidade em relação à classe mãe, a hierarquia de classes deve ser revista.

Classes abstratas e Interfaces

Classes abstratas são classes que podem conter tanto métodos abstratos (veremos mais adiante a definição), assim como métodos convencionais. Não podem ser instanciadas. Sua utilização exige o uso de subclasses que irão implementar os métodos abstratos.

Métodos abstratos são métodos que não possuem corpo (implementação). Contém apenas a declaração. Sua presença torna a classe automaticamente abstrata. Ou seja, se uma classe possui pelo menos um método abstrato, é considerada uma classe abstrata. Esses métodos abstratos devem ser implementados nas subclasses da classe abstrata que os definiu.

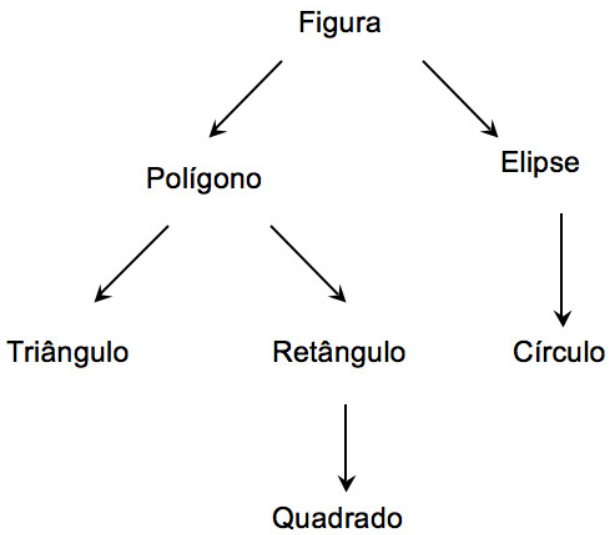
Fique atento!

Uma classe deve ser declarada abstrata se:

- A classe possui um ou mais métodos abstratos;
- A classe herda um ou mais métodos abstratos, os quais não são implementados;
- A classe declara que implementa uma determinada interface mas não fornece implementação para todos os métodos desta interface.

EXEMPLO

Considere uma hierarquia de classes envolvendo herança de figuras geométricas:



Fonte: Próprio Autor.

A classe Figura define uma figura genérica. Considere que cada subclasse dessa hierarquia definida por Figura contém um método desenhar ( ). Não é possível definir um método desenhar para uma figura genérica; este método deve ser abstrato, devendo ser implementado em cada subclasse de Figura.

A classe Figura poderia ser definida como abstrata. Veja como ficaria o código no exemplo abaixo:

```
public abstract class Figura {

    // método abstrato; subclasses de figura devem
    // fornecer uma implementação para esse método.
    public abstract void desenhar ( ) ;

}
```

São consideradas classes abstratas "puras" aquelas que possuem apenas métodos abstratos. Os atributos, quando presentes, se resumem a constantes estáticas. Podem ser definidas como interfaces, para maior flexibilidade no uso.

Interfaces são espécies de classes que possuem apenas métodos abstratos. Os atributos, se existirem, devem ser public, static e final. Não podem ser instanciadas. Sua utilização exige o uso de uma classe que implemente a interface.

Essas funcionam como um "contrato": Uma classe que implementa uma interface deve implementar todos os métodos dessa interface. Ao implementar uma interface, a classe mantém o direito de herdar de outra classe. Uma classe pode implementar diversas interfaces, incrementando o polimorfismo. Ficou curioso(a) para saber o que significa polimorfismo? Fique tranquilo, veremos polimorfismo na próxima unidade.

Classes que desejam implementar uma interface utilizam a palavra reservada implements.

EXEMPLO

Declaração da Interface:

```
public interface MinhaInterface  {

        public void meuMetodoAbstrato ( ) ;
}
```

Implementação de Interface em uma subclasse:

```
public class MinhaClasse implements MinhaInterface  {

        public void meuMetodoAbstrato ( ) {
                System.out.println ( " Método implementado ! " ) ;
        }
}
```

PALAVRAS DO PROFESSOR

Bem, caro(a) aluno(a)! Chegamos ao fim da unidade 3. Nela vimos alguns pilares usados na Orientação a Objetos. Aprendemos como usar o encapsulamento, criar classes usando herança e como criar classes abstratas e Interfaces.

Espero que tenha gostado deste material, qualquer dúvida entre em contato com seu tutor.

Aguardo você na próxima unidade. Até lá!