PROGRAMAÇÃO ORIENTADA À OBJECTOS

HERANÇA

INTRODUÇÃO

Suponha que pretendemos desenvolver uma aplicação para auxiliar nos trabalhos de uma determinada instituição que estuda as classes de alguns animais. Para esta instituição importa recolher algumas informações sobre as espécies das diferentes classes estudadas. As classes e os respectivos dados que se pretende recolher encontram-se listados abaixo.

- Insectos: nome, alimentação, habitat, e a informação de se o insecto é ou não venenoso;
- Peixes: nome, alimentação, habitat e tipo de barbatanas;
- Mamíferos: nome, alimentação, habitat e quantidade de membros de locomoção

Usando os conhecimentos que temos agora criaríamos 3 classes de objectos, a classe Insecto, Peixe e Mamífero. Nestas três classes teríamos 3 atributos em comum, nomeadamente: o nome, a alimentação e o habitat.

Vamos imaginar que depois de termos o nosso sistema funcional, fosse necessário acrescentar mais um atributo comum a todas as classes, digamos, a data de nascimento. Seria necessário incluir em todas as classes este novo atributo comum. Esta solução não é óptima, pois, obriga-nos a triplicar o esforço toda vez que quisermos inserir uma modificação comum a todas as classes do nosso sistema. Além disso, com esta solução, espalhamos códigos idênticos no nosso sistema o que tornará a manutenção dificultosa. Imagine que os possíveis habitats de todas as espécies das classes fossem predefinidos (por exemplo: vales, florestas ou riachos). Seria necessário colocar em cada uma das classes uma validação do habitat! E como seria no dia em que a forma de validar mudasse?! Seria necessário percorrer todas as classes e modificar o código que valida o habitat! Precisamos de uma solução melhor para resolver este problema.

E se fosse possível agrupar todos os atributos comuns numa classe e invocar esta classe nas restantes classes do sistema acrescentando nestas os atributos não comuns?! Essa solução serial ideal pois todos os atributos comuns estariam agrupados e o tratamento destes estaria centralizado! Em java é possível conseguir esse efeito! E isso consegue-se recorrendo a um mecanismo chamado Herança.

DEFINIÇÃO

Mecanismo pelo qual uma classe herda (uma parte ou todos) os atributos e métodos da classe da qual descende. Isto significa que, se uma classe descende doutra, ela possuirá os atributos e métodos da classe mãe (super classe). A classe que descende doutra é considerada classe filha (descendente/subclasse). Neste tipo de relacionamento, diz-se que a classe filha estende a classe Mãe.

O mecanismo de herança, aumenta os ganhos da programação orientada à objectos, uma vez que permite que classes escritas sejam reutilizadas sem necessidades de serem "remexidas".

A classe filha poderá acrescentar (a ela mesma) atributos e métodos, que não foram contemplados na classe Mãe; além disso, a classe filha poderá reescrever algumas funcionalidades da classe Mãe, isto é, implementar novamente funcionalidades definidas na classe Mãe.

Para estabelecer uma herança entre duas classes, usa-se a palavra reservada *extends*.

EXEMPLO 1

Vamos resolver o problema levantado no início deste artigo usando o mecanismo de herança. Primeiro vamos criar a classe mãe (a classe que contem todos os atributos comuns). Chamemos esta classe de *Animal*. Depois iremos criar as classes filhas e nestas vamos acrescentar os atributos a elas exclusivos.

```
1
2  public class Animal {
3      protected String nome;
4      protected String alimentacao;
5      protected String habitat;
6
7      public Animal(){}
8
9      public Animal(String nome, String alimentacao, String habitat){
10         this.nome = nome;
11         this.alimentacao = alimentacao;
12         this.habitat = habitat;
13     }
14
15     public String getNome() {return nome;}
16
17     public void setNome(String nome) {this.nome = nome;}
18
19     public String getAlimentacao() {return alimentacao;}
20
21     public void setAlimentacao(String alimentacao) {this.alimentacao = alimentacao;}
22
23     public String getHabitat() {return habitat;}
24
25     public void setHabitat(String habitat) {this.habitat = habitat;}
26 }
```

Fig 1. A classe animal (classe genérica ou classe mãe)

```java
public class Insecto extends Animal {
    private boolean venenoso;

    public Insecto(String nome, String alimentacao, String habitat, boolean venenoso){
        this.nome = nome;
        this.alimentacao = alimentacao;
        this.habitat = habitat;
        this.venenoso = venenoso;
    }

    public boolean isVenenoso() {
        return venenoso;
    }
    public void setVenenoso(boolean venenoso) {
        this.venenoso = venenoso;
    }
}
```

Fig 2. A classe Insecto

```java
public class Mamifero extends Animal {
    private int qtdMembros;

    public Mamifero(String nome, String alimentacao, String habitat, int qtdMembros){
        this.nome = nome;
        this.alimentacao = alimentacao;
        this.habitat = habitat;
        this.qtdMembros = qtdMembros;
    }

    public int getQtdMembros() {
        return qtdMembros;
    }

    public void setQtdMembros(int qtdMembros) {
        this.qtdMembros = qtdMembros;
    }
}
```

Fig 3. A classe Mamífero

```java
public class Peixe extends Animal {
    String tipoBarbatanas;

    public Peixe(String nome, String alimentacao, String habitat, String tipoBarbatanas){
        this.nome = nome;
        this.alimentacao = alimentacao;
        this.habitat = habitat;
        this.tipoBarbatanas = tipoBarbatanas;
    }

    public String getTipoBarbatanas() {return tipoBarbatanas;}
    public void setTipoBarbatanas(String tipoBarbatanas) {
        this.tipoBarbatanas = tipoBarbatanas;
    }
}
```

Fig 4. A classe Peixe

MANIPULAÇÃO DE OBJECTOS DE CLASSE COM HERANÇA

Dissemos que as classes filhas (subclasses) herdam os atributos e métodos da classe mãe (super classes). Isto significa que podemos manipular estes métodos e atributos como se eles tivesse sido definidos na classe filha.

EXEMPLO 2

Elabore um programa capaz de gerar dois objectos, um do tipo Mamífero e outro do tipo Insecto. Após a criação destes objectos visualize os valores dos seus atributos.

Resolução

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

        Insecto i = new Insecto("Aranha", "Folhas verdes", "Continente", true);

        Mamifero m = new Mamifero("Boi", "Capim verde", "Florestas", 4);

        System.out.println(i.getNome() + " "+ i.getHabitat() + " " + i.isVenenoso());

        System.out.println(m.getNome() +" "+ i.getAlimentacao() +" "+ m.getQtdMembros());
    }
}
```

Note que invocamos, sobre objectos do tipo Mamífero e Insecto, métodos definidos na classe Animal, como se estes tivessem sido definidos nestas classes (subclasses).

INVOCAÇÃO DE MÉTODOS DA SUPER CLASSE NA SUBCLASSE

Voltemos a nossa atenção aos construtores das nossas 4 classes. Repare que todos eles possuem três linhas idênticas, as linhas de inicialização dos atributos comuns. Será que havia necessidade de repetir estas linhas? A resposta é não! Se o construtor da super classe "sabe" inicializar as variáveis comuns, não há necessidade de fazermos esta inicialização nas classes filhas. O que deveríamos ter feito é invocar o construtor da super classe nas classes filhas. Isso é possível e é feito usando a palavra reservada *super* e a ele especificar os valores dos atributos do construtor. Repare como isso pode ser feito na subclasse Insecto.

```
1
2  public class Insecto extends Animal {
3      private boolean venenoso;
4
5      public Insecto(String nome, String alimentacao, String habitat, boolean venenoso){
6          super(nome, alimentacao, habitat);
7          this.venenoso = venenoso;
8      }
9
10     public boolean isVenenoso() {
11         return venenoso;
12     }
13     public void setVenenoso(boolean venenoso) {
14         this.venenoso = venenoso;
15     }
16 }
```

A palavra *super* é também usada quando queremos invocar métodos da super classe que usam identificadores iguais. Suponha por exemplo que na classe Animal, tivéssemos um método chamado *calculaIdade()* e na classe Mamífero tivéssemos um método com o mesmo nome. Para nos referirmos ao método criado na classe Animal teríamos de usar a palavra super a anteceder o chamamento ao método.

```
1
2  public class Animal {
3      protected String nome;
4      protected String alimentacao;
5      protected String habitat;
6
7      public Animal(){}
8
9      public Animal(String nome, String alimentacao, String habitat){
10         this.nome = nome;
11         this.alimentacao = alimentacao;
12         this.habitat = habitat;
13     }
14
15     public double calculaIdade(){
16         //Codigo para calcular a idade
17     }
18 }
```

```java

public class Insecto extends Animal {
    private boolean venenoso;

    public Insecto(String nome, String alimentacao, String habitat, boolean venenoso){
        super(nome, alimentacao, habitat);
        this.venenoso = venenoso;
    }

    pubic double calculaIdade(){
        double idadeSuper = super.calculaIdade();

        //Codigo calcula idade da subclasse
    }
}
```

POLIMORFISMO

Suponha que no problema anterior se faça necessário achar o tempo de vida dos animais, isto é, as idades. A solução seria incluir na super classe (Animal) o atributo data de nascimento, e incluir o método calcula idade para achar a idade do animal com base na data de nascimento. A nossa classe Animal ficaria assim:

```
import java.util.Date;

public class Animal {
    protected String nome;
    protected String alimentacao;
    protected String habitat;
    protected java.util.Date dataNascimento;

    public Animal(){}

    public Animal(String nome, String alimentacao, String habitat, Date data){
        this.nome = nome;
        this.alimentacao = alimentacao;
        this.habitat = habitat;
        this.dataNascimento = data;
    }

    public double calculaIdade(Date dataActual){
        double idade = DateHelper.dateDiff(dataActual, dataNascimento);

        return idade;
    }

    //gets e sets
}
```

Note que, o método ***calculaIdade***(), calcula o tempo de vida em anos. Note ainda que foi usando o método *dateDiff* da classe DateHelper para achar a diferença de datas. Esta não é uma classe de java, por tanto caso queira testar este exemplo terás de incluir a classe *Datehelper* no teu projecto. Está classe encontra-se no anexo deste artigo.

Vamos supor que a idade dos insectos fosse medida em dias e não em anos e que a idade dos peixes fosse medida em meses. Neste caso teríamos de reescrever o método calcula ***calculaIdade()*** para as subclasses Insecto e Peixe, de tal maneira que as idades estejam respectivamente em dias e em meses. As nossas classes Insecto e Peixe ficariam assim:

```java
import java.util.Date;

public class Insecto extends Animal {
    private boolean venenoso;

    public Insecto(String nome, String alimentacao, String habitat, boolean venenoso, Date data){
        super(nome, alimentacao, habitat, data);
        this.venenoso = venenoso;
    }

    public double calculaIdade(Date dataActual){
        double idadeEmAnos = super.calculaIdade(dataActual);
        double idadeEmDias = idadeEmAnos * 365;

        return idadeEmDias;
    }

    //gets e sets
}
```

```java
import java.util.Date;

public class Peixe extends Animal {
    String tipoBarbatanas;

    public Peixe(String nome, String alimentacao, String habitat, String tipoBarbatanas, Date data){
        super(nome, alimentacao, habitat, data);
        this.tipoBarbatanas = tipoBarbatanas;
    }

    public double calculaIdade(Date dataActual){
        double idadeEmAnos = super.calculaIdade(dataActual);

        double idadeEmMeses = idadeEmAnos * 12;

        return idadeEmMeses;
    }

    //gets e sets
}
```

O que fizemos foi reescrever o método ***calculaIdade()*** nas subclasses *Insecto* e Peixe.

EXEMPLO

Escreva um exemplo que mostra a invocação do método *calculaIdade*() das classes Mamífero, Insecto e Peixe e diga qual será o resultado dessa invocação.

Resolução

```
1  import java.util.Date;
2
3  public class TestaReescrita {
4      public static void main(String[] args){
5
6          Date dataActual = DateHelper.gerarData(2011, 1, 1, 0, 0);
7
8          Date dataNascimentoMamifero = DateHelper.gerarData(2010, 1, 1, 0, 0);
9          Date dataNascimentoInsecto = DateHelper.gerarData(2010, 1, 1, 0, 0);
10         Date dataNascimentoPeixe = DateHelper.gerarData(2010, 1, 1, 0, 0);
11
12         Mamifero mamifero = new Mamifero("Leao", "Carne Vermelha", "Florestas", 4, dataNascimentoMamifero);
13         Insecto insecto = new Insecto("Aranha", "Folhas verdes", "Continente", true, dataNascimentoInsecto);
14         Peixe peixe = new Peixe("Carapau", "Minhocas", "Agua doce", "Lisas", dataNascimentoPeixe);
15
16         System.out.println(mamifero.getNome() + " " + mamifero.calculaIdade(dataActual));
17         System.out.println(insecto.getNome() + " " + insecto.calculaIdade(dataActual));
18         System.out.println(peixe.getNome() + " " + peixe.calculaIdade(dataActual));
19     }
20 }
```

De acordo com a leitura dos dados dos objectos, nota-se que os chamamento do método *calculaIdade*() sobre o objecto **mamífero** irá retornar o valor 1 que corresponde a 1 ano; para o objecto insecto, o método retornará 365, o qual corresponde a 365 dias; finalmente, para o objecto peixe, o retorno será de 12, que corresponde a 12 meses. Procure justificar o porquê destes resultados.

Agora considere as linhas abaixo.

```
Date dataActual = DateHelper.gerarData(2011, 1, 1, 0, 0);

Date dataNascimentoPeixe = DateHelper.gerarData(2010, 1, 1, 0, 0);
Peixe peixe = new Peixe("Carapau", "Minhocas", "Agua doce", "Lisas", dataNascimentoPeixe);

Animal animal = peixe;

System.out.println(animal.getNome() + " " + animal.calculaIdade(dataActual));
```

Será que a atribuição feita na quarta instrução é correcta? Esta atribuição parece não estar correcta, pois estamos a fazer uma atribuição de um peixe a um animal (referenciados por duas variáveis de tipos diferentes). Mas será que um peixe não é um animal? O peixe é sim um animal! Logo esta atribuição é correcta. Nesta linha fizemos com que a variável animal referenciasse o objecto peixe, esta última referencia um objecto do tipo Peixe, logo, tanto a variável peixe como animal referenciam o mesmo objecto. Temos aqui o que se chama, em programação orientada à objectos, de Polimorfismo.

Polimorfismo é a capacidade de um objecto poder ser referenciado de várias formas. No nosso exemplo referimo-nos ao Peixe como Animal. Afinal de contas todo o peixe é um animal!

E qual será o resultado da invocação do método *calculaIdade*() na linha 5? O resultado será o retorno da idade em meses! Isso soa estranho pois estamos a invocar o método sobre uma variável do tipo Animal. Mas dissemos antes que esta variável faz referência a um objecto do tipo *Peixe,* logo, o método invocado será o definido nesta classe (Peixe).

O polimorfismo adiciona poder à programação orientada à objectos.

Considere que se pretende criar uma classe com métodos capazes de escrever dados de um animal num ficheiro de texto (nome, alimentação, habitat e idade); Os dados deverão ser escritos da mesma forma para qualquer tipo de animal, seja mamífero, peixe ou insecto MAS a idade deverá ser impressa no formato correspondente à classe do animal (anos para mamíferos, meses para peixes e dias para insectos). Usando o polimorfismos torna-se possível criar um único método para todas as extensões da classe animal. O método receberia pelo parâmetro uma referência de animal de qualquer um das classes de animais existentes. A classe e o método em causa ficariam como ilustrado na figura abaixo:

```
1  import java.io.BufferedWriter;
2  import java.io.FileWriter;
3  import java.io.IOException;
4  import java.util.Date;
5
6  public class DataBaseAnimal {
7
8      public static void main(String[] args) throws IOException{
9          Date dataActual = DateHelper.gerarData(2011, 1, 1, 0, 0);
10
11         Date dataNascimentoPeixe = DateHelper.gerarData(2010, 1, 1, 0, 0);
12         Date dataNascimentoInsecto = DateHelper.gerarData(2010, 1, 1, 0, 0);
13
14
15         Peixe peixe = new Peixe("Carapau", "Minhocas", "Agua doce", "Lisas", dataNascimentoPeixe);
16
17         Insecto insecto = new Insecto("Mosquito", "Sangue vivo", "Charcos", true, dataNascimentoInsecto);
18
19         writeAnimal(insecto, dataActual);
20         writeAnimal(peixe, dataActual);
21     }
22
23     public static void writeAnimal(Animal a, Date data) throws IOException{
24         FileWriter file = new FileWriter("info.txt", true);
25         BufferedWriter buffer = new BufferedWriter(file);
26
27         String linha = a.getNome() +"<>"+ a.alimentacao + "<>"+ a.getHabitat() +"<>"+ a.calculaIdade(data);
28
29         buffer.write(linha);
30         buffer.newLine();
31         buffer.close();
32         file.close();
33     }
34 }
```

Repare que no nosso *main* temos duas invocações do método *writeAnimal*! Por este método passa-se as referências de objectos do tipo *Insecto e Peixe*, respectivamente. Do lado do método *writeAnimal* o método *calculaIdade* será invocado sobre a classe correspondente à referência e não sobre a classe Animal; logo, o efeito será o desejado: a escrita da idade no formato correspondente à classe do Animal. Sem polimorfismo seria necessário criar tantos métodos *writeAnimal* quanto o número de classes filhas, pois, o parâmetro de cada um destes seria diferente; mas graças ao Polimorfismo torna-se possível fazer referência aos objectos de todas as classes filhas usando a super classe. Esta é uma vantagem do polimorfismo!

CLASSES ABSTRACTAS

Graças ao polimorfismo podemos fazer referência aos objectos das subclasses com a super classe. No exemplo anterior, nos referimos a mamífero, peixe ou insecto como Animal; isso nos trouxe um ganho significativo, pois com um único método (*writeAnimal)* foi possível responder a um problema para as várias subclasses da classe Animal. Voltemos a nossa atenção ao problema levantado no início deste artigo. Com que fim criamos a classe Animal? O objectivo era agrupar as características comuns às classes de animais Mamífero, Insecto e Peixe, que são classe que importa estudar; logo, a classe Animal é uma classe auxiliar que foi criada justamente para contorna um problema, o problema de repetição de código. Há bocado vimos que esta classe também ajuda-nos a conseguir polimorfismo. Mas será que em algum momento na nossa aplicação haverá necessidade de criar um objecto do tipo Animal? A resposta é "não!" pois, o que está em estudo são as três classes: Mamífero, Insecto e Peixe. Logo não faria sentido nenhum a criação de uma instância da classe Animal. O java possui um mecanismo para inibir a criação de instância de uma classe. Isso é conseguido definindo a classe com Abstracta.

Uma classe abstracta é uma classe que não pode ser instanciada, ela funciona como um template de todas as suas subclasses. A classe abstracta é definida usando a palavra-chave *abstract,* conforme ilustrado abaixo.

```
import java.util.Date;

 abstract class Animal {
    protected String nome;
    protected String alimentacao;
    protected String habitat;
    protected java.util.Date dataNascimento;

    public double calculaIdade(Date dataActual){
        double idade = DateHelper.dateDiff(dataActual, dataNascimento);

        return idade;
    }
}
```

A partir deste momento, torna-se impossível criar um objecto do tipo *Animal!* Qualquer tentativa de instanciar o Animal irá gerar um erro de compilação.

Mas qual é a importância de classes abstractas? A grande vantagem de classes abstractas é a possibilidade de adicionar métodos abstractos nelas. Um método abstracto é nada mais que uma simples declaração de método sem nenhuma implementação. A implementação deverá ser feita em cada uma das classes filhas da classe em causa. E qual é vantagem disso? Voltemos ao nosso exemplo. A nossa aplicação possui três classes e elas reescreveram o método calculaIdade de tal maneiras que a idade esteja em concordância com a forma de contagem da idade da classe (anos, para mamíferos; meses, para peixes e dias para insectos). O que aconteceria se se criasse uma nova classe, digamos, Verme, cujo tempo de vida mede-se em horas. Se o criador desta classe se esquecer de reescrever o método calculaIdade, erradamente o cálculo da idade será herdado da classe Animal o que gerará resultados errados pois, o calculaidade da classe Animal, calcula a idade em anos e não em horas. Quando um método é definido como abstracto, o compilador obriga a implementação deste em todas as suas classes filhas. Este efeito evita o esquecimento de reescrita de métodos (conforme aconteceu com Verme) por parte dos programadores que fazem a extensão de classes com métodos que precisam de ser reescritos.

Um método abstracto é definido apenas com a parte da assinatura do método, sem nenhum corpo. Abaixo define-se o método calculaIdade como abstracto.

```
import java.util.Date;

abstract class Animal {
   protected String nome;
   protected String alimentacao;
   protected String habitat;
   protected java.util.Date dataNascimento;

   abstract double calculaIdade(Date dataActual);
}
```

A partir de já, qualquer classe que estender a classe Animal deverá obrigatoriamente implementar o método *calculaIdade.*

EXERCÍCIOS

1. Suponha que foste convidado para participar de um projecto de desenvolvimento de uma aplicação bancária para um determinado banco. Segundo o levantamento feito, existe no banco dois tipos de contas: contas a ordem e contas a prazo. Todas as contas possuem um número, titular e saldo. As contas a ordem possuem, para além dos atributos comuns, o regime de titularidade o qual pode ser cada um destes três: individual, solidária, conjunta ou mista. Toda vez que se efectua um depósito numa conta a prazo, o saldo é acrescido com 3% do valor do depósito. Já para as contas a ordem, quando um depósito for efectuado é cobrado uma comissão de 1% do valor da transacção. Usando os conceitos de herança, sugira um modelo de classes para as contas deste sistema. (As classes sugeridas deverão possuir atributos e respectivos métodos).
2. Com base nas classes criadas no exercício anterior, crie um programa que permita criar uma conta a ordem e outra a prazo. Depois da criação das contas efectue o seguinte:
    a. Deposite 1000,00 em cada uma das contas.
    b. Visualize o saldo de cada uma das contas. (A que se deve a diferença dos saldos?)
    c. Referencie cada uma das contas com a super classe e efectue novamente um depósito de 1000,00 em cada uma das contas. Que efeitos se espera para o saldo de cada uma das contas?
    d. Na alínea anterior pode se dizer que aplicou-se polimorfismo? Justifique.

3. Suponha que se faça necessário saber o ***saldo real*** de uma determinada conta, isto é, o saldo sem juros ou comissões. Para as contas a ordem, o saldo real é calculado com a fórmula seguinte: **saldo real = saldo + (saldo*1) /100**. E para as contas a prazo, **saldo real = saldo - (saldo*3) /100**.

    Altere as tuas classes de tal maneiras que seja possível achar o saldo real das contas.

    Após alterar as contas, crie uma classe que possua um método (*escreveNoFicheiro*) que permite escrever os dados de uma conta independentemente do seu tipo. No ficheiro, deverá ser escrito, o número da conta, o saldo e o saldo real. Adicione nesta classe o main e ilustre a aplicação do método *escreNoFicheiro.*

4. Sem dúvida, no modelo de classes criado no exercício 1, criaste uma super classe. Agora, altere a super classe de tal maneiras que esta seja abstracta. Nela, deverás incluir os seguintes métodos abstractos: *deposita* (método que efectua o depósito) e *calculaSaldoReal* (método que calcula o saldo real). De seguida faça as modificações necessárias nas classes filhas.

ANEXO

A classe DateHelper

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

public class DateHelper {

    /**
    * Acha a diferenca em anos das datas passadas pelos parametros
    */
    public static double dateDiff(Date dataMaior, Date dataMenor){

        double diferencaEmMilissegundos =  dataMaior.getTime() - dataMenor.getTime();
        double diferencaEmSegundos =  diferencaEmMilissegundos/1000;
        double diferencaEmMinutos =  diferencaEmSegundos/60;
        double diferencaEmHoras =  diferencaEmMinutos/60;
        double diferencaEmDias =  diferencaEmHoras/24;

        double diferencaEmAnos =  diferencaEmDias/365;

        return diferencaEmAnos;
    }

    public static Date gerarData(int ano, int mes, int dia, int hora, int minutos){
        return new Date(ano-1900, mes-1, dia, hora, minutos);
    }
}
```