Disciplina de Programação Orientada a Objetos - POOS3

Curso Superior de ADS - 3º Semestre

(Professor Dênis Leonardo Zaniro)

Classes abstratas e interfaces

Sumário

- Classes abstratas e concretas
- A classe Object
- Interfaces e polimorfismo

Objetivos

1- Aprender a utilizar classes abstratas, e entender sua importância no desenvolvimento de aplicações flexíveis.

2- Entender a utilidade da classe Object.

3- Aprender a utilizar interfaces e entender as diferenças entre interfaces e classes abstratas.

Classes abstratas e concretas

No exemplo dado na aula anterior relacionado à hierarquia de animais, leões, hipopótamos, tigres, cachorros, lobos e gatos poderiam ser instanciados usando-se o operador *new*, conforme mostra o exemplo de código dado abaixo:

```
Leao leao = new Leao();
Tigre tigre = new Tigre(); //essa lógica se aplica a todas as outras subclasses
```

Da mesma forma que é possível instanciar objetos das subclasses de animais, também é possível instanciar um objeto da superclasse Animal, conforme é mostrado pela instrução dada abaixo:

```
Animal animal = new Animal();
```

Embora seja possível, do ponto de vista da compilação, instanciar objetos do tipo Animal, não faz sentido criar animais que não pertençam a algum tipo específico. Percebe-se que a classe é muito genérica para possuir objetos que contenham valores significativos. Para o trecho de código anterior, quais seriam os valores das variáveis de instância para um objeto do tipo Animal? Como seriam sua forma, cor, tamanho, etc.?

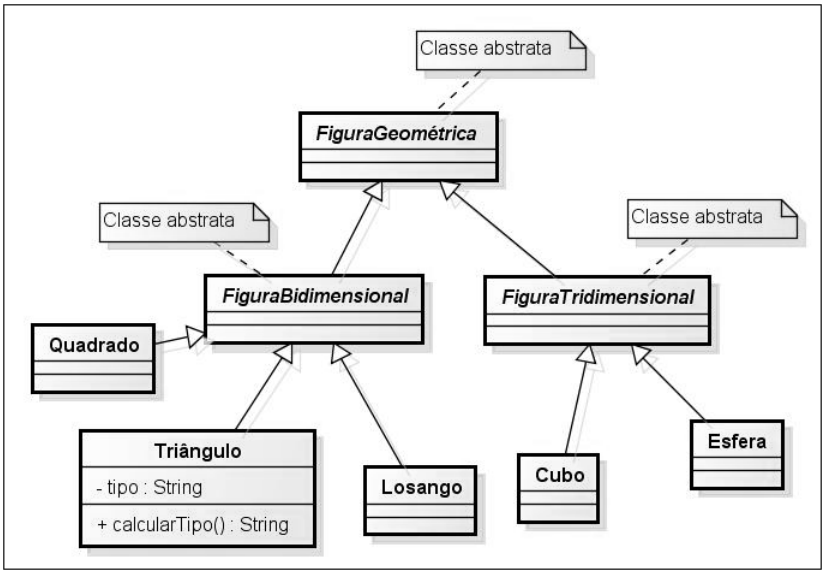
Nesse caso (e em muitos outros, como será estudado), deveremos usar um mecanismo que impeça que uma determinada classe seja instanciada. Para conseguirmos isso, deveremos declarar a classe como abstrata usando-se o modificador ***abstract***.

Ao marcar uma classe como *abstract*, o compilador não permitirá que sejam criadas instâncias diretas dessa classe em qualquer parte do código. O trecho de código abaixo exemplifica a declaração de uma classe abstrata:

```
public abstract class Animal {
  private String comida;
  private int fome;
  private int coordenadaX;
  private int coordenadaY;
  ...
}
```

Percebe-se, portanto, que, ao projetar uma estrutura de classes, deveremos decidir quais classes serão abstratas e quais classes serão concretas. Diferentemente das classes abstratas, classes concretas podem ser instanciadas diretamente. Até o momento, todas as classes criadas nos programas eram concretas.

Para o exemplo de figuras geométricas, também abordado na aula anterior, o mais correto seria declarar as classes FiguraGeométrica, FiguraBidimensional e FiguraTridimensional como abstratas. Dessa forma, pode-se garantir que não haverá instâncias diretas dessas três classes, uma vez que essas classes não representam figuras específicas o suficiente para serem instanciadas. O diagrama de classes abaixo mostra as três classes citadas especificadas como abstratas. Como pode ser observado no diagrama, nomes de classes abstratas devem ser escritos em itálico no padrão da UML.



É importante observar que classes abstratas somente possuem algum sentido se forem estendidas. Na API Java, existem várias classes abstratas como, por exemplo, as classes Component e Number. A classe Component é a superclasse das classes relacionadas às GUIs (*Graphical User Interfaces*), tais como botões, caixas de diálogo, caixas de texto, barras de rolagem e vários outros componentes gráficos. Ao criar componentes, não

criaremos objetos do tipo Component, mas objetos de classes que estendem a classe Component.

Já a classe Number é a superclasse de classes tais como Integer, Double, Float, e outras classes que encapsulam valores numéricos. Novamente, não haverá objetos do tipo Number, mas objetos do tipo Integer, do tipo Double, e assim por diante. Em outras palavras, será possível instanciar apenas objetos das subclasses concretas da classe abstrata Number.

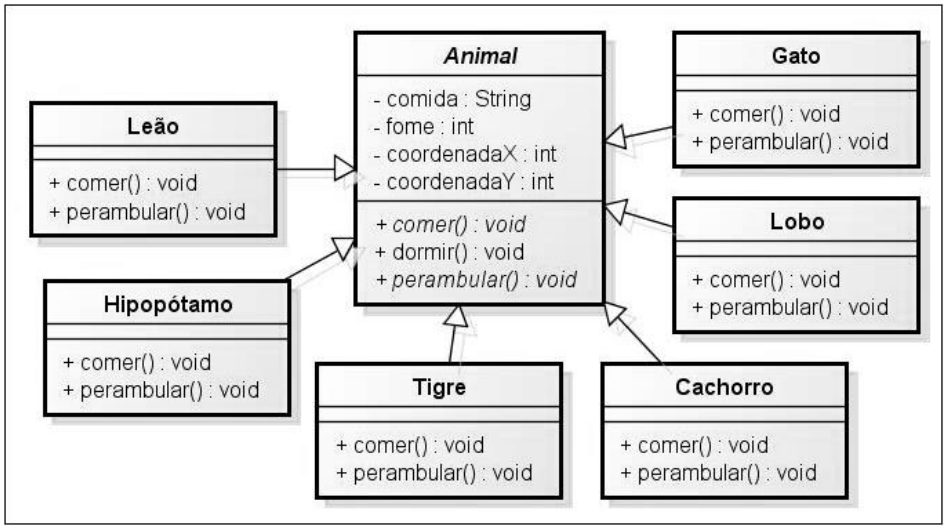
Ao declarar uma classe como abstrata, será possível definir métodos que também sejam abstratos. Nesse ponto, é importante observar que uma classe abstrata pode ter tanto métodos abstratos como métodos não-abstratos, diferentemente das classes concretas, nas quais somente podem ser definidos métodos não-abstratos.

Um método abstrato é simplesmente um método que não possui corpo. Portanto, declarar um método como abstrato significa estabelecer que esse método deverá ser sobreposto pelas classes que herdam da classe abstrata, onde o método abstrato foi declarado. O trecho de código abaixo mostra a declaração de um método abstrato:

```
public abstract class FiguraGeometrica {
  public abstract double calcularArea();
}
```

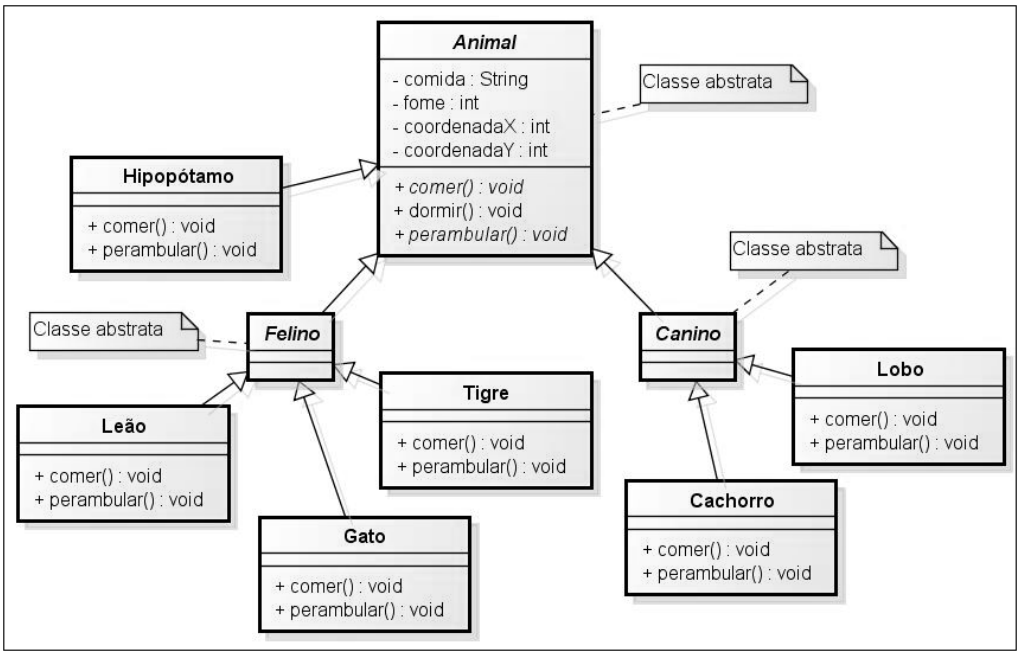
De acordo com o trecho de código anterior, não foi definido corpo algum para o método *calcularArea()*, isto é, não faria sentido definir um método para calcular a área de uma figura geométrica qualquer. Cada figura geométrica específica possui seu próprio cálculo da área, portanto, o correto é que esse método seja implementado por uma subclasse mais específica.

Considerando o domínio dos animais estudado na aula anterior, podemos perceber que os métodos *comer()* e *perambular()* também não deveriam possuir corpo na classe Animal, uma vez que cada tipo específico de animal possui sua própria forma de comer e perambular. A figura a seguir mostra o diagrama de classes modificado para o domínio de animais.



De acordo com o diagrama de classes apresentado anteriormente, todas as subclasses da classe Animal se obrigam a fornecer implementação aos métodos abstratos. Essa obrigação ocorre por que as subclasses de Animal são concretas. Caso houvesse uma subclasse abstrata, essa subclasse não teria mais a obrigação de implementar qualquer método abstrato definido na classe Animal. Essa obrigação seria delegada para classes concretas que herdassem da subclasse abstrata.

Como exemplo, vamos supor que identificamos, para esse domínio de animais, dois novos agrupamentos: um representando todos os animais felinos (leões, gatos e tigres), e outro representando todos os animais caninos (lobos e cachorros). O diagrama de classes dado abaixo ilustra essa situação:



De acordo com o diagrama, as classes Felino e Canino foram declaradas como classes abstratas, e, como tais, não possuem a obrigatoriedade de implementar os métodos abstratos definidos na classe Animal. Assim, essa obrigatoriedade foi delegada para as subclasses concretas das classes Felino e Canino.

É importante ressaltar que, caso uma classe precise definir, no mínimo, um método abstrato, então, essa classe deverá ser obrigatoriamente declarada como abstrata também.

A motivação para a definição de métodos abstratos está ligada diretamente ao uso do polimorfismo. Havendo métodos abstratos na superclasse, será possível usar uma referência do tipo da superclasse abstrata para invocar métodos implementados nas subclasses mais específicas. Dessa forma, poderemos acrescentar novos tipos a um programa sem precisar escrever novos métodos para lidar com esses tipos. O exemplo a seguir demonstra o uso do **polimorfismo**:

```
public abstract class Animal {
  ...
  public abstract void comer();
```

```
  public abstract void perambular();
  ...
}
public class Leao extends Animal {
  ...
  public void comer() {
    //forma de comer do leão
  }
  public void perambular() {
    //forma de perambular do leão
  }
  ...
}
public class Principal {
  public static void main(String args[]) {
    Animal a;
    a = new Leao();
    a.comer(); //chamada de comer() da classe Leao
    a.perambular(); //chamada de perambular() da classe Leao
  }
}
```

No código anterior, a instrução *a.comer()* invocará o método *comer()* definido na classe Leao, uma vez que o objeto real apontado pela variável *a* é do tipo Leao. Entretanto, para que essa instrução funcione do ponto de vista da compilação, é necessário que o método *comer()* também exista na classe Animal (como um método abstrato ou não).

Ainda observando o exemplo de código anterior, foi realizada uma conversão implícita de um tipo inferior (subclasse) para um tipo superior (superclasse). Caso seja necessário converter um tipo superior para um tipo inferior (subclasse), deveremos efetuar uma conversão explícita de tipos, conforme mostra o exemplo abaixo:

```
public class Conversao {
  public void converteTipos() {
    Animal a;
    Leao l;
    a = new Leao();
    l = (Leao) a; //operador cast usado para fazer a conversão
  }
}
```

Em muitas situações, antes de efetuarmos uma conversão explícita, precisaremos nos certificar de que uma determinada referência para a superclasse realmente aponta para objetos de uma determinada subclasse. Para realizar essa verificação (em tempo de execução), podemos utilizar o operador ***instanceof***. O trecho de código dado a seguir exemplifica o uso desse operador:

```
public class Conversao {
  public void converteTipos() {
    Animal a;
    a = new Leao();
    if(a instanceof Leao) //condição verdadeira
      System.out.println("a é um tipo de Leao!");
    a = new Tigre();
    if(a instanceof Leao) //condição falsa
      System.out.println("a é um tipo de Leao!");
    if(a instanceof Animal) //condição verdadeira
      System.out.println("a é um tipo de Animal!");
    Tigre t;
    t = new Tigre();
    if(t instanceof Tigre) //condição verdadeira
      System.out.println("t é um tipo de Tigre!");
    if(t instanceof Animal) //condição verdadeira
      System.out.println("t é um tipo de Animal!");
  }
}
```

De acordo com o código anterior, para usarmos o operador *instanceof*, devemos especificar uma referência de um determinado tipo antes desse operador e o nome do tipo (classe) após esse operador. Além disso, o operador *instanceof* somente pode ser usado para comparar tipos em uma mesma hierarquia de herança.

A classe Object

A classe Object é a classe mãe de todas as classes em Java. Portanto, todas as classes herdam direta ou indiretamente da classe Object. Se não houvesse essa classe, não haveria forma de os programadores criarem classes compostas de métodos genéricos capazes de lidar com quaisquer tipos criados em outros programas.

Por exemplo, o método *showMessageDialog()* da classe JOptionPane aceita como argumento o tipo Object, o que significa que qualquer tipo pode ser informado como argumento para esse método. Como resultado, objetos de qualquer tipo podem se beneficiar dos serviços da classe JOptionPane. É importante observar que essa regra se aplica a várias outras classes da API Java.

A classe Object implementa alguns comportamentos que são desejáveis para todo objeto em Java. São eles:

a) Método *equals()*: permite verificar se duas referências de um tipo qualquer apontam para o mesmo objeto.

b) Método *getClass()*: retorna a classe que representa o tipo de um determinado objeto.

c) Método *hashCode()*: retorna um código que identifica um determinado objeto.

d) Método *toString()*: retorna a representação de um objeto na forma de uma *string*.

Alguns métodos tais como *equals(), hashCode()* e *toString()* podem ser sobrepostos; outros métodos como o *getClass()* não podem ser sobrepostos - marcados com o modificador *final* - porque executam tarefas que não podem ser modificadas.

Embora a classe Object possa ser usada como um tipo genérico para tipos de retorno e parâmetros de métodos, é importante considerar que uma referência para o tipo Object somente poderá ser usada para invocar métodos que também constem na classe Object. Isso significa que, para acessarmos métodos específicos de um objeto real (Por exemplo, um tipo Integer ou um tipo String) apontado por uma referência do tipo Object, será necessário efetuar uma conversão. Além disso, como uma referência do tipo Object é capaz de armazenar qualquer tipo de objeto, não haverá garantia de que a conversão seja realizada com sucesso, uma vez que poderíamos esperar um Integer, mas obter, ao invés, uma String (Isso geraria uma exceção chamada *ClassCastException*).

Interfaces e polimorfismo

Vamos supor que, a partir desse momento, as classes Cachorro e Gato tivessem que ser reutilizadas em um programa para gerenciar animais domésticos - um programa para um *Pet Shop*. Esse programa deve lidar com animais domésticos de forma que não haja a necessidade de se conhecer detalhes específicos dos animais domésticos. Sabe-se que todo animal doméstico precisa saber brincar - método *brincar()* e ser amigável - método *serAmigavel()*.

Portanto, esses métodos precisarão ser inseridos de alguma forma na hierarquia de classes projetada inicialmente para o domínio de animais. Há, basicamente, três opções a se considerar, inicialmente:

1) *Inserir os métodos na classe Animal*: se especificássemos os métodos na classe Animal, todas as subclasses, automaticamente, passariam a "enxergar" esses métodos. Além disso, o programa apenas precisaria conhecer detalhes da classe Animal, e qualquer classe que estendesse futuramente a classe Animal também poderia se beneficiar dos métodos herdados. O problema dessa abordagem é que animais como leões (classe Leão), tigres (classe Tigre) e outros animais selvagens também passariam a herdar comportamentos que são característicos apenas de animais domésticos.

2) *Inserir os métodos abstratos na classe Animal*: se declarássemos os métodos na classe Animal como abstratos, o problema de animais selvagens herdarem

comportamentos de animais domésticos seria resolvido. Cada subclasse concreta, nesse caso, deveria implementar seus métodos e, para o caso dos animais selvagens, os métodos *brincar()* e *serAmigavel()* poderiam ser tornados inúteis. O problema dessa abordagem é que não faria sentido "obrigar", pelo contrato definido, as classes que representam animais selvagens a implementar métodos que não correspondem aos seus comportamentos.

3) *Inserir os métodos apenas nas classes Cachorro e Gato*: nessa abordagem, os problemas das outras duas opções seriam resolvidos, uma vez que apenas as classes Cachorro e Gato implementariam os métodos inerentes a animais domésticos. O novo problema, nesse caso, é que não haveria contrato algum definido, e, sempre que novas classes fossem adicionadas ao programa, seria de responsabilidade dos programadores implementar os métodos *brincar()* e *serAmigavel()* em suas respectivas classes. Além disso, não seria possível usar o recurso do polimorfismo, e qualquer classe que precisasse lidar com animais domésticos precisaria ser modificada sempre que um novo animal tivesse que ser considerado pelo programa.

A "solução" para esse caso seria criar uma nova classe, chamada AnimalDomestico que fosse composta de todos os métodos relacionados a animais domésticos. Dessa classe, herdariam as classes Cachorro e Gato, além de outras classes que fossem acrescentadas futuramente ao programa. Entretanto a linguagem Java não suporta herança múltipla, portanto, as classes Cachorro e Gato não podem ter duas superclasses diretas.

Diante dessa situação, o único recurso em Java que poderia ser utilizado refere-se às **interfaces**. Devemos pensar na interface como uma "classe" especial que é composta somente de métodos públicos abstratos. Não existe implementação em uma interface, mas apenas declarações de métodos abstratos. Por esse motivo, costuma-se dizer que uma interface é uma classe abstrata pura. O exemplo de código abaixo mostra a declaração de uma interface em Java:

```
public interface AnimalDomestico {
  public abstract void brincar();
  public abstract void serAmigavel();
}
```

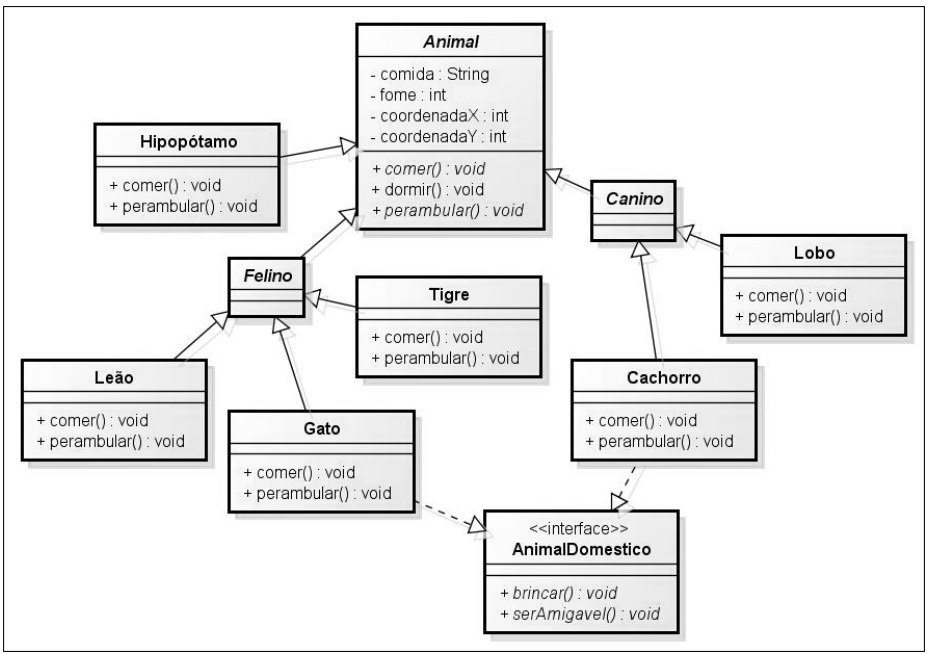
De acordo com o código anterior, uma interface em Java é criada usando-se a palavra-chave ***interface***. Nesse exemplo, declaramos dois métodos abstratos, o que significa que esses métodos deverão ser implementados por alguma classe. Para indicarmos que uma classe deverá implementar os métodos abstratos definidos em uma interface, será necessário usar a palavra-chave ***implements*** na definição da classe, conforme é mostrado pelo exemplo a seguir:

```
public class Cachorro extends Animal implements AnimalDomestico
{
  ...
  public void brincar() {
    ... //código específico para o cachorro
```

```
  }
  public void serAmigavel() {
    ... //código específico para o cachorro
  }
} //a mesma regra se aplica à classe Gato
```

Nesse exemplo, declaramos que a classe Cachorro estende a classe Animal e implementa a interface AnimalDomestico, isto é, um cachorro deve ter e fazer tudo que um animal qualquer faz, mas também deve fazer tudo que um animal doméstico faz. Em termos de código, quando uma classe estende outra classe e implementa uma interface, a palavra-chave *extends* deve preceder a palavra-chave *implements*.

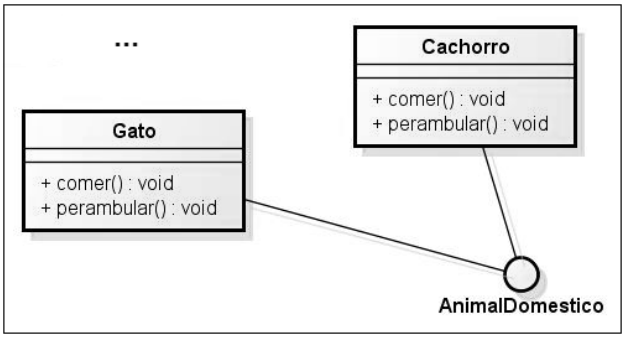
É importante observar que, uma classe, ao implementar uma interface, declara que aceita o contrato estipulado por ela, portanto assumirá a responsabilidade de implementar todos os métodos definidos na interface. O diagrama de classes da UML dado a seguir demonstra o relacionamento entre duas classes e uma interface.



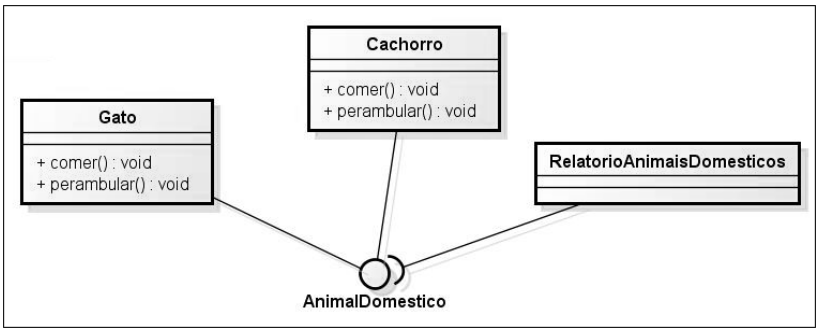
Pelo diagrama anterior, percebe-se que existe uma linha tracejada partindo das classes Gato e Cachorro com uma seta apontando para a interface AnimalDomestico. Essa notação gráfica significa que esse é um relacionamento do tipo *realization* em UML, isto é, um relacionamento que indica que uma determinada classe implementa uma interface.

O elemento <<*interface*>> que aparece sobre o nome da interface é um ***estereótipo*** que denota que AnimalDomestico é uma interface. Estereótipos são mecanismos que a UML fornece que permitem usar um conceito já existente na linguagem adaptando-o para acomodar uma nova situação. Nesse caso, usamos a notação de classe para representar uma interface por meio de um estereótipo. O mecanismo de estereótipo é um recurso poderoso que permite estender a capacidade de modelagem da UML.

Além de usar estereótipos, é possível representar interfaces graficamente. O diagrama de classes dado a seguir mostra que as classes Gato e Cachorro implementam a interface AnimalDomestico. Costumamos dizer que essas duas classes fornecem a interface AnimalDomestico obedecendo às suas restrições. As interfaces fornecidas são representadas por um círculo fechado ligado à classe por uma linha sólida.



Considerando-se ainda o programa para gerenciar animais domésticos em um Pet Shop, caso houvesse uma classe para lidar com animais domésticos, essa classe precisaria conhecer apenas a interface AnimalDomestico. Vamos supor que essa classe precisasse gerar um relatório de todos os animais domésticos considerados pelo programa. Poderíamos criar uma classe RelatorioAnimaisDomesticos que dependeria da interface AnimalDomestico. O diagrama de classes abaixo mostra graficamente esse relacionamento de dependência de uma classe em relação à interface AnimalDomestico.



Quando uma classe depende de uma interface, costumamos dizer que a interface é requerida pela classe. Interfaces requeridas são representadas por um semicírculo ligado a uma classe por uma linha sólida.

Como já foi mencionado, uma interface somente possui métodos abstratos, assim, os modificadores *public* e *abstract* podem ser omitidos da declaração de métodos em uma interface. Dessa forma, o código da interface dado a seguir é semelhante ao código apresentado anteriormente:

```
public interface AnimalDomestico {
  void brincar();
  void serAmigavel();
} //implicitamente, são métodos públicos e abstratos
```

Além de possuir métodos abstratos, interfaces também podem possuir constantes estáticas públicas. Essas constantes são úteis para armazenar dados que serão

compartilhados por todas as classes que implementarem a interface. Portanto, qualquer campo declarado em uma interface deverá ser dos tipos *static* e *final*, e qualquer classe que implementar a interface terá acesso a esses campos. O trecho de código dado abaixo exemplifica a declaração de uma constante estática na interface AnimalDomestico:

```
public interface AnimalDomestico {
  public static final int X = 10;
  void brincar();
  void serAmigavel();
}
public class Cachorro extends Animal implements AnimalDomestico
{
  public void imprimeX() {
    System.out.println(X);
  }
}
```

É importante observar que quaisquer campos declarados em uma interface serão, por *default*, constantes estáticas públicas, assim, o código da interface do exemplo anterior poderia ser substituído pelo código dado a seguir:

```
public interface AnimalDomestico {
  int X = 10; //constante estática pública
  void brincar();
  void serAmigavel();
}
```

Embora uma classe possa estender apenas uma superclasse diretamente, é possível que uma classe implemente mais de uma interface ao mesmo tempo. Suponha que as classes Cachorro e Gato tivessem que ser reutilizadas no domínio de uma clínica veterinária representando animais pacientes na clínica. Dessa forma, deveria haver mais um método a ser inserido nessas classes: o método *consultar()*.

Seguindo a mesma lógica usada para criar a interface AnimalDomestico, uma nova interface seria criada - interface AnimalPaciente - e as classes Cachorro e Gato se responsabilizariam em implementar também essa interface. A seguir, são apresentados os códigos referentes à nova interface e à classe Cachorro (a mesma regra observada para a classe Cachorro se aplica à classe Gato).

```
public interface AnimalPaciente {
  void consultar();
}
public class Cachorro extends Animal implements AnimalDomestico,
AnimalPaciente {
  public void brincar() {
```

```
    ... //método descrito na interface AnimalDomestico
  }
  public void serAmigavel() {
    ... //método descrito na interface AnimalDomestico
  }
  public void consultar() {
    ... //método descrito na interface AnimalPaciente
  }
}
```

Pelo código dado acima, percebe-se que, se uma classe concreta se propuser a implementar duas ou mais interfaces, os métodos de todas as interfaces deverão ser obrigatoriamente implementados. As interfaces implementadas por uma classe devem ser separadas por vírgula na sua declaração, conforme é mostrado no código.

Classes abstratas também podem implementar interfaces, entretanto, por serem abstratas, essas classes não terão a obrigatoriedade de implementar os métodos declarados na interface. É possível que alguns dos métodos declarados na interface sejam implementados pela classe abstrata e, caso isso aconteça, as subclasses concretas da classe abstrata não terão mais a responsabilidade de implementar esses métodos. Nesse caso, as subclasses concretas terão apenas a responsabilidade de implementar os métodos que não foram implementados pela classe abstrata.

É possível que uma interface estenda outra interface, usando-se a palavra-chave ***extends***. Uma interface B que estenda uma interface A passará a incorporar todos os métodos abstratos e constantes estáticas declaradas na interface A, e poderá definir novos métodos abstratos e constantes estáticas.

Nesse caso, uma classe concreta que implemente a interface B assumirá a responsabilidade de implementar todos os métodos abstratos declarados na interface B e todos os métodos abstratos que a interface B herdou da interface A. O exemplo dado abaixo mostra a herança entre as interfaces AnimalPaciente e AnimalDomestico.

```
public interface AnimalPaciente extends AnimalDomestico {
  void consultar();
}
```

De acordo com esse exemplo, a interface AnimalPaciente passou a incorporar os métodos abstratos da interface AnimalDomestico, isto é, definiu-se que, qualquer classe, ao implementar a interface AnimalPaciente, se obrigará a implementar também os métodos abstratos que constam na interface AnimalDomestico. Diante disso, para o exemplo da classe Cachorro, não seria mais necessário declarar explicitamente que essa classe implementa duas interfaces, mas apenas a interface AnimalPaciente. O exemplo abaixo mostra como deveria ficar a nova declaração da classe Cachorro:

```
public class Cachorro extends Animal implements AnimalPaciente {
  public void brincar() {
```

```
    ... //método descrito na interface AnimalDomestico
  }
  public void serAmigavel() {
    ... //método descrito na interface AnimalDomestico
  }
  public void consultar() {
    ... //método descrito na interface AnimalPaciente
  }
} //a mesma regra se aplica à classe Gato
```

Uma consideração importante sobre interfaces é que uma interface somente pode estender outras interfaces (nunca uma classe). Além disso, entre interfaces, pode haver herança múltipla, isto é, uma interface pode estender duas ou mais interfaces diretamente. O trecho de código dado abaixo representa o relacionamento de herança entre as interfaces A, B e C. Como pode ser observado, a interface C estende diretamente as interfaces A e B.

```
public interface A {
  ...
}
public interface B {
  ...
}
public interface C extends A,B {
  ...
}
```

É fundamental ressaltar que interfaces não são apenas usadas para permitir simular a herança múltipla, mas também para determinar um contrato para objetos de tipos diferentes, mas que possuam as mesmas operações.

Da mesma forma como acontece com classes abstratas, é possível usar uma referência do tipo de uma interface para acessar métodos fornecidos (que foram declarados na interface) por classes que implementam a interface. O exemplo de código abaixo demonstra esse mecanismo supondo que a classe Cachorro, por exemplo, implemente a interface AnimalDomestico (exemplos anteriores):

```
public class Principal {
  public static void main(String args[]) {
    AnimalDomestico i;
    i = new Cachorro();
    i.brincar(); //chamada polimórfica de brincar()
    i.serAmigavel(); //chamada polimórfica de serAmigavel()
```

```
  }
} //em ambas as chamadas, serão os métodos da classe Cachorro que serão executados.
```

Ao estudar classes abstratas e interfaces, uma questão importante normalmente levantada é sobre quando usar classes abstratas e quando usar interfaces. Classes abstratas devem ser criadas quando precisarmos definir um modelo para um grupo de subclasses e houver, pelo menos, algum código de implementação que seja compartilhado entre todas as subclasses. Já as interfaces devem ser usadas quando precisarmos definir operações em comum que outras classes possam desempenhar (implementar), independentemente da hierarquia de herança envolvendo essas classes. Em outras palavras, para usarmos interfaces, não é necessário que o relacionamento É UM se aplique.

Referências bibliográficas

Deitel, H. M.; Deitel, P. J. Java - Como Programar. 4. ed., Bookman, 2002.

Oracle. *Learning the Java Language*. Disponível em: http://download.oracle.com/javase/tutorial/java/. Data de acesso: 02/01/2016.

Santos, R. Introdução à Programação Orientada a Objetos usando JAVA. Elsevier, 2003.

Sierra, K; Bates, B. Use a Cabeça! Java. 2. ed., O Reilly, 2005.