Disciplina de Programação Orientada a Objetos - POOS3

Curso Superior de ADS - 3º Semestre

(Professor Dênis Leonardo Zaniro)

Herança e Introdução ao Polimorfismo

Sumário

- Herança entre classes
- Sobreposição de métodos
- O modificador *final*
- O modificador de acesso *protected*
- Polimorfismo

Objetivos

1- Entender o conceito de herança na análise orientada a objetos e aplicar esse conceito na programação em Java.

2- Explorar os benefícios que a herança proporciona, e, principalmente, entender quais são as situações às quais a herança se aplica.

3- Entender o que é polimorfismo e como esse conceito auxilia na elaboração de códigos mais facilmente extensíveis.

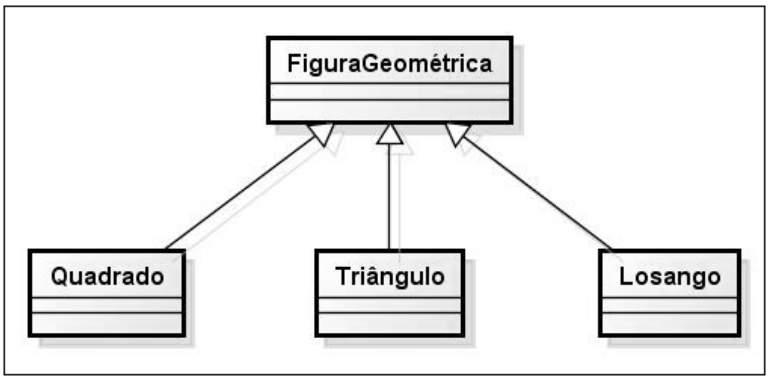
Herança entre classes

Além dos relacionamentos entre classes já estudados, existe o relacionamento de **herança** (também conhecido como especialização/generalização). A herança entre classes ocorre quando uma classe - frequentemente chamada de *subclasse* - herda dados e comportamentos de outra classe - frequentemente chamada de *superclasse*. Costumamos dizer que a subclasse *estende* a superclasse. Além de herdar, uma subclasse pode incorporar novas características - variáveis de instância e métodos - que dizem respeito apenas a ela.

A herança favorece a reutilização de código, uma vez que ela permite evitar a duplicação de código nas subclasses. O relacionamento de herança é um relacionamento do tipo "É UM". Portanto, se uma classe B estende uma classe A, pode-se dizer que a classe B é uma classe A. O relacionamento "É UM" se aplica apenas em uma direção, assim é correto afirmar, por exemplo, que chá é uma bebida, mas não podemos dizer que bebida é um chá (nem todas as bebidas são chás).

Vamos supor que seja necessário desenvolver um programa que gerencie diversas figuras geométricas diferentes. Identificando algumas figuras geométricas, percebemos que todas elas compartilham características, portanto, poderemos inserir essas

características em uma classe que as outras classes de figuras possam estender. Bastará apenas informar às classes mais específicas que a classe mais abstrata será a sua superclasse. O exemplo abaixo mostra o relacionamento de herança entre classes que representam figuras geométricas.



De acordo com o diagrama apresentado anteriormente, foi criada uma classe chamada FiguraGeométrica - a superclasse - que contém as características comuns que todas as figuras possuem. Dessa forma, é possível que outras classes que representem figuras específicas - as subclasses - possam estender a classe FiguraGeométrica. Nesse diagrama, as subclasses consideradas são Quadrado, Triângulo e Losango, entretanto, qualquer outra classe que também represente uma determinada figura geométrica poderia ser acrescentada a essa hierarquia posteriormente. É importante observar que a superclasse não saberá da existência de subclasses.

O código em Java que permite traduzir o diagrama de classes anterior é apresentado a seguir:

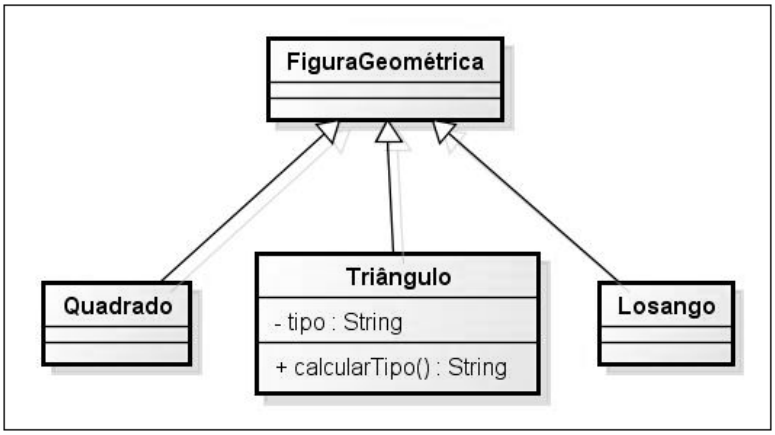
```
public class FiguraGeometrica {
   ...
}
public class Quadrado extends FiguraGeometrica {
   ...
}
public class Triangulo extends FiguraGeometrica {
   ...
}
```

De acordo com o código anterior, para indicarmos a existência de herança entre classes, deveremos usar a palavra-chave ***extends***. Se uma classe denominada *B* ***extends*** uma classe denominada *A*, então a classe *B* herda tudo que existe na classe *A*, e ainda poderá incorporar novas características, como já foi mencionado.

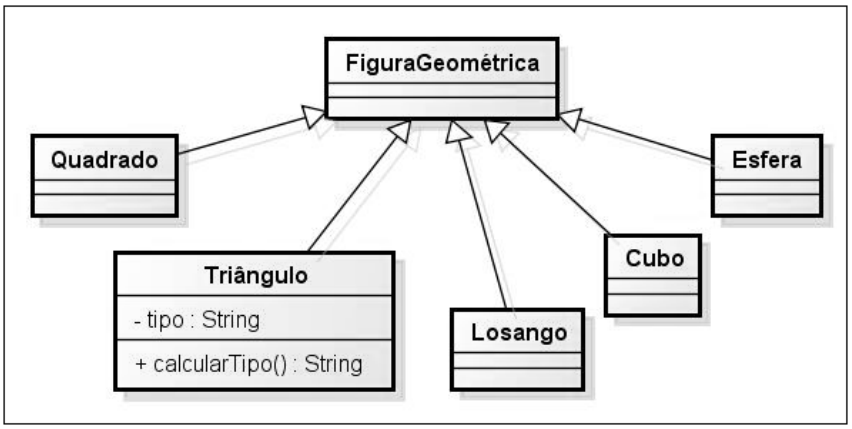
Se uma classe denominada *C* herda de uma classe *B*, e a classe *B*, por sua vez, herda de uma classe *A*, então, indiretamente, a classe *C* passará a herdar da classe *A*. Da mesma forma, se uma classe *D* herdar da classe *C*, então a classe *D* passará a herdar das classes *A*, *B* e *C*, automaticamente.

A linguagem Java não suporta herança múltipla, portanto, uma classe não pode herdar diretamente de duas ou mais superclasses. Em outras palavras, para cada subclasse, somente pode haver uma superclasse direta.

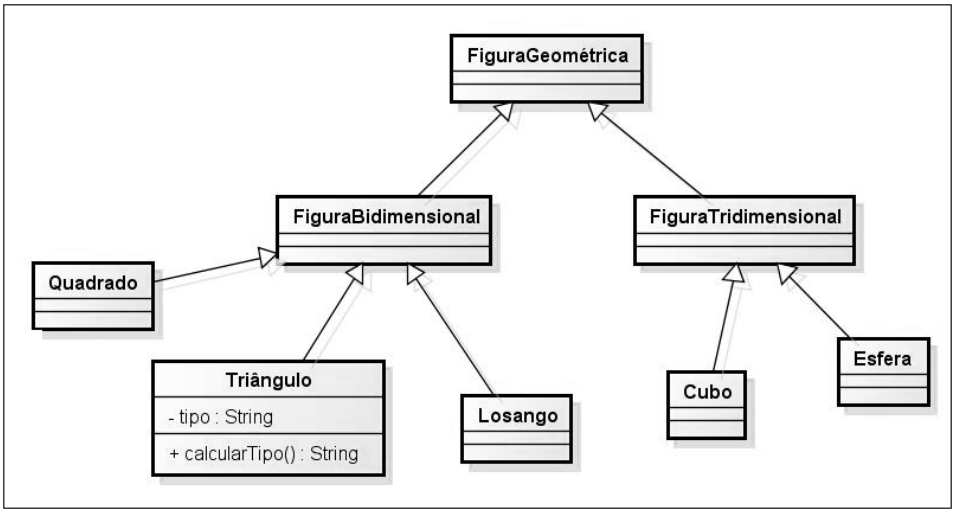
Ainda no exemplo das figuras geométricas, percebemos que um triângulo, além de possuir tudo que uma figura geométrica possui, também possui um tipo (“equilátero”, “isósceles” ou “escaleno”) e uma operação para calcular o seu tipo. Como essas características são específicas do triângulo, elas deverão ser adicionadas à classe Triângulo, conforme é mostrado pelo diagrama abaixo.



Vamos supor que fosse necessário trabalhar com mais figuras geométricas tais como cubos e esferas. Como cubos e esferas também são figuras geométricas (satisfazendo o princípio “É UM”), poderemos acrescentar as suas classes correspondentes a essa hierarquia, conforme mostra o diagrama abaixo:



Após acrescentar essas figuras, podemos perceber que quadrado, triângulo e losango são figuras bidimensionais, e, como tais, possuem características comuns. O mesmo ocorre com cubos e esferas, uma vez que são figuras tridimensionais. Dessa forma, é possível ainda melhorar essa hierarquia criando-se dois novos agrupamentos: um grupo que represente as figuras bidimensionais e outro grupo que represente as figuras tridimensionais. O diagrama apresentado a seguir ilustra esse novo cenário.



De acordo com o diagrama anterior, quadrados, triângulos e losangos são figuras bidimensionais e, como figuras bidimensionais são figuras geométricas, então, concluímos que quadrado, triângulo e losango são figuras geométricas. O mesmo se aplica a cubos e esferas - figuras tridimensionais. Essa conclusão, embora seja intuitiva, precisa ser explicitada para percebermos que a classe FiguraGeometrica é a superclasse indireta de todas as classes de figuras específicas.

O trecho de código para o diagrama anterior é dado abaixo:

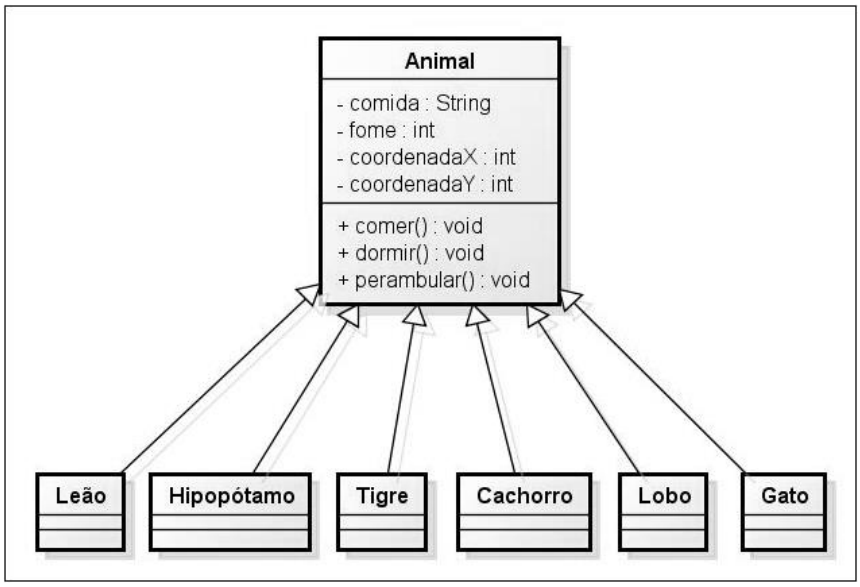
```
public class FiguraGeometrica {
  ...
} //classe mãe de todas as classes de figuras
public class FiguraBidimensional extends FiguraGeometrica {
  ...
} //classe mãe das classes de figuras bidimensionais
public class FiguraTridimensional extends FiguraGeometrica {
  ...
} //classe mãe das classes de figuras tridimensionais
public class Quadrado extends FiguraBidimensional {
  ...
} //classe específica
public class Triangulo extends FiguraBidimensional {
  ...
} //classe específica
public class Losango extends FiguraBidimensional {
  ...
} //classe específica
public class Cubo extends FiguraTridimensional {
  ...
```

```
} //classe específica
public class Esfera extends FiguraTridimensional {
  ...
} //classe específica
```

Movendo-se do domínio de figuras geométricas, vamos supor que seja necessário, a partir desse momento, desenvolver um programa para gerenciar diversas espécies de animais. Inicialmente (e este número poderá ser aumentado), identificamos seis tipos de animais, e cada animal deverá ser representado por um objeto diferente.

Após identificarmos esses tipos, percebemos que todos esses animais possuem um tipo de comida preferido (atributo *comida*), um nível de fome (atributo *fome*) e coordenadas *x* e *y* que representam a posição do animal no espaço. Além disso, todos os animais se alimentam (método *comer*), dormem (método *dormir*) e circulam pelo espaço (método *perambular*).

Assim, todas essas características comuns foram reunidas em uma classe denominada Animal, da qual todas as outras classes que representam tipos diferentes de animais herdam. O diagrama de classes dado abaixo ilustra essa situação.



A partir desse diagrama, suponha que, para criar qualquer animal, o seu tipo de comida e o seu nível de fome devam ser obrigatoriamente informados. Nesse caso, deverá haver apenas um construtor (com parâmetros) na classe Animal que receba esses valores para inicializar um animal qualquer.

Havendo apenas um construtor com parâmetros na classe Animal (superclasse), todas as suas subclasses deverão também possuir um construtor que invoque explicitamente o construtor da classe Animal. Para invocar o construtor da classe animal, deveremos usar a instrução ***super***. Além disso, essa instrução, quando usada, deve ser a primeira instrução do construtor da subclasse. O trecho de código dado as seguir mostra essa situação, considerando-se apenas as subclasses Leao e Hipopotamo (A mesma lógica se aplica às outras classes de animais).

```
public class Animal {
  private String comida;
  private int fome;
  private int coordenadaX;
  private int coordenadaY;
  public Animal(String comida, int fome) {
    this.comida = comida;
    this.fome = fome;
  }
  public void comer() {
    ...
  }
  public void dormir() {
    ...
  }
  public void perambular() {
    ...
  }
}
public class Leao extends Animal {
  public Leao(String comida, int fome) {
    super(comida,fome); //chamada do construtor da classe Animal
  }
  ...
}
public class Hipopotamo extends Animal {
  public Hipopotamo(String comida, int fome) {
    super(comida,fome); //chamada do construtor da classe Animal
  }
  ...
}
```

Pelo código especificado acima, percebemos que a chamada ao construtor da superclasse se tornou obrigatória, uma vez que existe somente um construtor com parâmetros na superclasse. Se, nessa superclasse, houvesse, além do construtor com parâmetros, um construtor *default* (sem parâmetros), o uso da instrução *super* não seria obrigatório em nenhuma das subclasses. Nesse caso, a instrução *super* somente deveria ser usada se a subclasse necessitasse invocar o construtor com parâmetros. Portanto, considere o uso da instrução *super* nas seguintes situações:

a) A superclasse define apenas construtores que recebam parâmetros, isto é, não há construtor *default*. Nesse caso, a instrução *super* é obrigatória em todas as subclasses para invocar algum construtor da superclasse.

b) A superclasse define construtores com parâmetros e o construtor *default*. Nesse caso, a instrução *super* não é obrigatória, entretanto, caso alguma subclasse precise invocar algum construtor com parâmetro da superclasse, a instrução *super* deverá ser usada.

c) A superclasse define apenas o construtor *default* ou não define construtores. Em qualquer uma dessas situações, não é necessário usar a instrução *super*. É importante observar que essa instrução ainda pode ser usada da seguinte forma: *super()*.

O código abaixo ilustra uma das situações mencionadas.

```
public class Animal {
  ...
  public Animal(String comida, int fome) {
    this.comida = comida;
    this.fome = fome;
  }
  public Animal() { }
  ...
}
public class Leao extends Animal {
  public Leao(String comida, int fome) {
    super(comida,fome); //chamada do construtor com parâmetros
  }
  public Leao() { } //chamada automática do construtor default
  ...
} //a mesma lógica se aplica a todas as outras subclasses
```

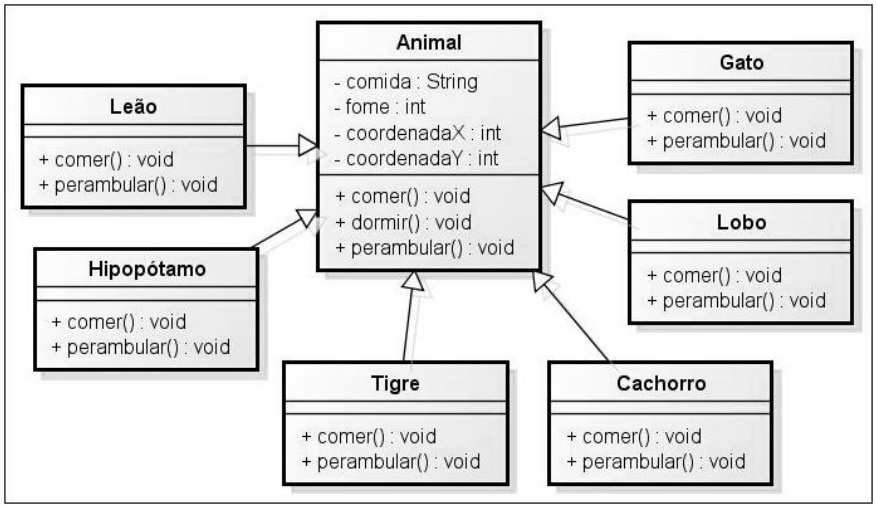
De acordo com o código acima, a superclasse Animal definiu dois construtores - um com dois parâmetros e outro *default*, portanto, as subclasses podem decidir em chamar o construtor com parâmetros ou o construtor *default*. Em outras palavras, qualquer espécie de animal pode ser criada a partir de valores iniciais definidos para a comida e a fome ou sem quaisquer valores. No caso da classe Leao, foi definido um construtor com parâmetros que deverá invocar o construtor com parâmetros da classe Animal e o construtor *default* (que não precisa invocar qualquer construtor da superclasse).

Sobreposição de métodos

Considerando-se ainda o exemplo dos animais, é sensato imaginar que qualquer animal possua um tipo de alimento, um nível de fome, uma posição no espaço, além de comer,

dormir e circular. Entretanto, não seria sensato imaginar que todos os animais comem ou circulam exatamente da mesma forma.

Para resolver esse problema, as subclasses podem sobrepor comportamentos que são inerentes a elas. A **sobreposição** (ou sobrescrita) de métodos implica em implementar, em uma classe, uma nova versão de um método já existente em uma superclasse. É importante observar que a operação continuará sendo a mesma, mas a sua implementação variará de subclasse para subclasse. O diagrama de classes a seguir mostra a sobreposição de métodos para os tipos de animais.



De acordo com esse diagrama, percebe-se que cada tipo de animal definiu sua própria versão dos métodos *comer( )* e *perambular( )*, isto é, cada tipo de animal possui diferenças em sua forma de comer e perambular pelo espaço. O trecho de código a seguir mostra essa sobreposição de métodos.

```
public class Animal {
  ...
  public void comer() { ... }
  public void perambular() { ... }
  ...
}
public class Leao extends Animal {
  ...
  public void comer() {
    ... //método comer() sobreposto para um leão
  }
  public void perambular() {
    ... //método perambular() sobreposto para um leão
  }
  ...
} //a mesma regra se aplica a todos os outros animais
```

Como pode ser observado no código acima, para sobrepor um método, é necessário que a nova versão implementada tenha o mesmo tipo de retorno e defina os mesmos tipos de parâmetros (se houver). Além disso, o modificador de acesso deve ser o mesmo ou então menos restritivo.

Por exemplo, supondo que as classes Animal e Leao do exemplo estejam no mesmo pacote, seria possível haver um método *comer( )* na classe Animal sem modificador de acesso (modificador *default*), e implementar o método *comer( )* na classe Leao com o modificador *public*. O trecho de código a seguir mostra essa situação:

```
public class Animal {
  ...
  void comer() {
    ... //método comer() com o modificador default
  }
  ...
}
public class Leao extends Animal {
  ...
  public void comer() {
    ... //método comer() sobreposto com o modificador public
  }
  ...
}
```

Considerando-se esse exemplo de código, caso uma referência para um objeto do tipo Leao seja usada para invocar o método *comer( )*, o método *comer( )* sobreposto na classe Leao será invocado. Como regra, sempre que usarmos uma referência para um objeto para chamar algum método, chamaremos, na verdade, a versão mais específica desse método para esse tipo de objeto. O exemplo de código abaixo mostra uma classe que instancia um objeto Leao e, a partir desse objeto, invoca o método *comer( )*.

```
public class Principal {
  public static void main(String args[]) {
    Leao leao = new Leao();
    leao.comer(); //chamada do método comer() da classe Leao
  }
  ...
}
```

É importante observar que, se definirmos um método que possua o mesmo nome de outro método implementado em uma superclasse, mas com parâmetros diferentes, haverá uma sobrecarga de métodos (e não sobreposição). O exemplo de código a seguir

demonstra o que aconteceria, caso o método *comer( )* fosse definido na classe Leao usando-se um parâmetro.

```
public class Animal {
  ...
  public void comer() {
    ...
  }
  ...
}
public class Leao extends Animal {
  ...
  public void comer(int quantidade) {
    ...
  }
  ...
}
public class Principal {
  public static void main(String args[]) {
    Leao leao = new Leao();
    leao.comer(10); //chamada do método comer() da classe Leao
    leao.comer(); //chamada do método comer() da classe Animal
  }
  ...
}
```

No exemplo de código anterior, percebemos que, caso uma referência para o objeto Leao seja usada para invocar o método *comer( )* com um argumento inteiro, será o método *comer( )* com parâmetro (da classe Leao) que será executado. Caso nenhum argumento seja informado, o método *comer( )* sem parâmetro, herdado da classe Animal, será executado.

Considerando-se ainda o método *comer( )* que qualquer animal possui, suponha que antes de um animal comer de fato, algumas tarefas precisassem ser realizadas, e essas tarefas fossem iguais para todos os animais. Diante dessa situação, para não ter que repetir essas tarefas em cada subclasse, poderíamos inseri-las no método *comer( )* da classe Animal.

Assim, antes de o método *comer( )* ser executado para um animal específico, o método *comer( )* da superclasse deveria ser chamado. Em outras palavras, antes de executar o método sobreposto na subclasse, a versão desse método na superclasse deveria ser chamada. Para que um método qualquer definido em uma subclasse invoque um método

definido na superclasse, temos que usar a palavra-chave ***super*** precedendo o nome do método. O exemplo de código abaixo ilustra o uso da instrução ***super*** nesse contexto.

```
public class Animal {
  ...
  public void comer() {
    ...
  }
  ...
}
public class Leao extends Animal {
  ...
  public void comer() {
    super.comer(); //chamada do método comer() da classe Animal
    ...  //código específico para o leão
  }
  ...
}
```

É importante observar que não há sobreposição de métodos estáticos. É possível que uma subclasse defina um método estático que possua a mesma assinatura de outro método estático implementado na superclasse. Entretanto, não é permitido que uma subclasse defina um método não-estático com a mesma assinatura de um método estático já existente em uma superclasse.

O modificador *final*

Ao definirmos uma classe qualquer em Java, é possível estabelecer que um ou alguns dos métodos nessa classe não poderão ser sobrepostos, isto é, não será possível definir uma nova versão desse método. Para impedir que um método seja sobreposto, deveremos usar o modificador ***final***. O exemplo de código abaixo mostra o uso desse modificador na classe Animal, apresentada em exemplos anteriores.

```
public class Animal {
  ...
  public final void perambular() {
    ...
  } //esse método não poderá ser sobreposto
  ...
}
```

No exemplo acima, ao definir o método *perambular( )* como *final*, nenhuma subclasse de animal poderá sobrepor esse método. Normalmente, isso ocorre quando é necessário preservar a funcionalidade original de um dado método contido em uma classe. Além de

definir métodos *final*, é possível ainda definir uma classe inteira como *final*. Uma classe do tipo *final* não pode ser herdada por qualquer classe. Caso a classe Animal fosse do tipo *final*, nenhuma outra classe de animal poderia estendê-la. Várias classes da API Java são declaradas com o modificador *final* (Por exemplo, a classe String).

O modificador de acesso *protected*

Além dos modificadores *public* e *private* já estudados, ao aplicar o conceito de herança, podemos usar o modificador *protected* para quaisquer membros de uma classe. Um membro de uma classe que possua o modificador *protected* poderá ser acessado diretamente pela própria classe onde ele foi declarado, por todas as subclasses e também pelas classes que estão no mesmo pacote da classe que o contém. O trecho de código dado a seguir exemplifica o uso do modificador *protected*.

```
package br.edu.ifsp.ads.poo;
public class Animal {
  protected String comida;
  ...
}
package br.edu.ifsp;
public class Leao extends Animal {
  public void setComida(String comida) {
    this.comida = comida;
  } //a classe Leao tem acesso direto ao atributo herdado comida
  ...
}
package br.edu.ifsp.ads.poo; //pacote da classe Animal
public class Principal {
  public static void main(String[] args) {
    Leao leao = new Leao();
    leao.comida = "carne";
  } //essa classe também tem acesso direto ao atributo comida
}
```

Polimorfismo

Pelo o que foi observado até o momento, um dos principais benefícios da herança é que ela permite evitar duplicação de código. Além disso, ao definir uma superclasse, definiremos, na verdade, um protocolo garantindo que as subclasses tenham todos os métodos que a superclasse tem.

Com base nisso, objetos de uma subclasse poderão ser usados em qualquer ponto onde objetos da superclasse seriam usados. Em outras palavras, podemos projetar códigos para manipular objetos das superclasses, e esses códigos serão capazes de funcionar

também para objetos das subclasses. É justamente esse mecanismo que permitirá trabalhar com o ***polimorfismo***.

Uma variável de referência, por exemplo, declarada para objetos da superclasse poderá referenciar objetos de uma determinada subclasse. Considerando o exemplo de animais, o trecho de código dado abaixo mostra esse mecanismo.

```
public class Principal {
  public static void main(String[] args) {
    Animal animal; //referência para objetos do tipo Animal
    animal = new Leao(); //atribuição de um objeto leao para a referência do tipo Animal
    animal.comer(); //chamada ao método comer() da classe Leao
  }
}
```

De acordo com o trecho de código acima, uma referência para um objeto do tipo Animal (superclasse) está sendo usada para controlar um objeto do tipo Leao (subclasse). A partir disso, o método *comer( )* implementado na classe Leao será chamado usando-se a referência para a superclasse Animal. Essa chamada de método somente é possível por que a superclasse Animal também define um método *comer( )*, mesmo que esse método não faça sentido nessa classe (Nas próximas aulas, veremos como resolver esse problema).

Além de atribuir objetos de uma subclasse para uma referência para superclasse, é possível haver parâmetros e tipos de retorno polimórficos. O trecho de código abaixo mostra um método de uma classe, cujo parâmetro é uma referência para a superclasse Animal.

```
public class GerenciaAnimal {
  public void alimentaAnimal(Animal animal) {
    animal.comer();  //chamada ao método comer() do objeto enviado como argumento
  }
}
public class Principal {
  public static void main(String[] args) {
    Leao leao = new Leao();
    Cachorro cachorro = new Cachorro();
    GerenciaAnimal ga = new GerenciaAnimal();
    ga.alimentaAnimal(leao); //leão é um animal
    ga.alimentaAnimal(cachorro);  //cachorro é um animal
  }
}
```

De acordo com o código anterior, dentro do método *alimentaAnimal( )*, a chamada *animal.comer( )* invocará o método *comer( )* da classe cujo objeto foi informado como argumento. Observe que qualquer objeto que seja de uma classe que estenda a classe Animal poderá ser informado como argumento.

Na classe Principal, é criado um objeto do tipo Leao e outro objeto do tipo Cachorro, e ambos podem ser informados como argumento para o método *alimentaAnimal( )*, uma vez que eles são animais. De acordo com o exemplo, primeiramente é o método *comer()* da classe Leão que será executado, e, em um segundo momento, é o método *comer()* da classe Cachorro que será executado.

Portanto, a decisão sobre qual método *comer()* será chamado dependerá do animal informado como argumento (essa decisão acontecerá em tempo de execução). Observe que a classe GerenciaAnimal não precisa conhecer especificamente quem é o animal para chamar o método *comer()* correspondente. Em outras palavras, essa classe é capaz de alimentar qualquer animal, sem que haja a necessidade de saber quem é o animal específico que está sendo alimentado.

Percebe-se que, com o polimorfismo, é possível escrever códigos que não precisem ser modificados se novos tipos de subclasses forem introduzidos ao programa. Nesse exemplo, caso outros animais passem a ser considerados pelo programa (a partir de novas classes representando animais), a classe GerenciaAnimal continuará funcionando sem que haja a necessidade de modificações.

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.