Sobrecarga, Herança e Polimorfismo

Aula 08

Gustavo Willam Pereira

Introdução

- A construção de classes, embora de fundamental importância, não representa o mecanismo mais importante da orientação à objetos. A grande contribuição da orientação à objetos para o projeto e desenvolvimento de sistemas é o polimorfismo.
- A palavra polimorfismo vem do grego *poli morfos* e significa muitas formas. Na orientação à objetos representa uma característica onde se admite tratamento idêntico para formas diferentes baseado em relações de semelhança, isto é entidades diferentes podem ser tratadas de forma semelhante conferindo grande versatilidade aos programas e classes que se beneficiam destas características.

Sobrecarga de Métodos

- A forma mais simples de polimorfismo oferecido pela linguagem Java é a sobrecarga de métodos (*method overload*) ou seja é a possibilidade de existirem numa mesma classe vários métodos com o mesmo nome.
- Para que estes métodos de mesmo nome possam ser distinguidos eles devem possuir uma assinatura diferente. A assinatura (*signature*) de um método é uma lista que indica os tipos de todos os seus argumentos, sendo assim métodos com mesmo nome são considerados diferentes se recebem um diferente número ou tipo de argumentos e tem, portanto, uma assinatura diferente.
- Um método que não recebe argumentos tem como assinatura o tipo *void* enquanto um outro método que recebe dois inteiros como argumentos tem como assinatura os tipos *int*, *int* como no exemplo a seguir:

Sobrecarga de Métodos

```
// Sobrecarga.java
public class Sobrecarga
{
    public long twice (int x)
     {
        return 2*(long)x;
    }
    public long twice (long x)
    {
        return 2*x;
    }
    public long twice (String x)
    {
        return 2*(long)Integer.parseInt(x);
    }
}
```

Sobrecarga de Métodos

- Outra situação possível é a implementação de métodos com mesmo nome e diferentes listas de argumentos, possibilitando uma utilização mais flexível das idéias centrais contidas nestes métodos como a seguir:

```
// Sobrecarga2.java
public class Sobrecarga2
{
	public long somatorio (int max) ]
	{
		int total=0;
		for (int i=1; i<=max; i++)
		total += i;
		return total;
	}
	public long somatorio (int max, int incr)
	{
		int total=0;
		for (int i=1; i<=max; i += incr)
		total += i;
		return total;
	}
}
```

Herança

- A herança (*inheritance*) é o segundo e mais importante mecanismo do polimorfismo e pode ser entendido de diversas formas das quais a mais simples é:
  - uma técnica onde uma classe passa a utilizar atributos e operações definidas em uma outra classe especificada como seu ancestral.
- Rigorosamente falando, a herança é o compartilhamento de atributos e operações entre classes baseado num relacionamento hierárquico do tipo pai e filho, ou seja, a classe pai contêm definições que podem ser utilizadas nas classes definidas como filho.
  - A classe pai é o que se denomina classe base (*base class*) ou superclasse (*superclass*) e as classes filho são chamadas de classes derivadas (*derived classes*) ou subclasses (*subclasses*).
  - Este mecanismos sugere que uma classe poderia ser definida em termos mais genéricos ou amplos e depois refinada sucessivamente em uma ou mais subclasses específicas. Daí o origem do termo técnico que descreve a herança: especialização (*specialization*).

Herança

- Em Java indicamos que uma classe é derivada de uma outra classe utilizando a palavra reservada *extends* conforme o trecho simplificado de código dado a seguir:
- A superclasse não recebe qualquer indicação especial.

```
// Arquivo: SuperClass.java
public class SuperClass
{
}

// Arquivo: SubClass.java
public class SubClass extends SuperClass
{
}
```

Herança

- Em princípio, todos atributos e operações definidos para uma certa classe base são aplicáveis para seus descendentes que, por sua vez, não podem omitir ou suprimir tais características pois não seriam verdadeiros descendentes se fizessem isto.
- Por outro lado uma subclasse (um descendente de uma classe base) pode modificar a implementação de alguma operação (reimplementar) por questões de eficiência sem modificar a interface externa da classe.
- Além disso as subclasses podem adicionar novos métodos e atributos não existentes na classe base, criando uma versão mais específica da classe base, isto é, especializando-a.

Herança

- A herança é portanto um mecanismo de especialização pois uma dada subclasse possui tudo aquilo definido pela sua superclasse além de atributos e operações localmente adicionados.
- No mundo real alguns objetos e classes podem ser descritos como casos especiais, especializações ou generalizações de outros objetos e classes.
  - Por exemplo, a bola de futebol de salão é uma especialização da classe bola de futebol que por sua vez é uma especialização da classe bola. Aquilo que foi descrito para uma certa classe não precisa ser repetido para uma classe mais especializada originada na primeira.

Herança

- Atributos e operações definidos na classe base não precisam ser repetidos numa classe derivada, desta forma a orientação à objetos auxilia a reduzir a repetição de código dentro de um programa ao mesmo tempo que possibilita que classes mais genéricas sejam reaproveitadas em outros projetos (reusabilidade de código).
- Enquanto mecanismo de especialização, a herança nos oferece uma relação “é um” entre classes (*“is a” relationship*): uma bola de futebol de salão é uma bola de futebol que por sua vez é uma bola.
- A descoberta de relações do tipo “é um” é a chave para determinarmos quando novas classes devem ser ou não derivadas de outras existentes.

Herança

- O acesso à atributos e operações declarados em uma classe base por parte das suas classes derivadas não é irrestrito. Tal acessibilidade depende dos nível de acesso permitido através dos especificadores *private* (privado), *protected* (protegido), e *public* (público)

Métodos e Atributos da Classe	Implemen-tação da Classe	Instância da Classe	SubClasse da Classe	Instância da SubClasse
privados (*private*)	sim	não	não	não
protegidos (*protected*)	sim	não	sim	não
públicos (*public*)	sim	sim	sim	sim

Herança

- Ao especificarmos acesso privado (*private*) indicamos que o atributo ou operação da classe é de uso interno e exclusivo da implementação da própria classe, não pertencendo a sua interface de usuário tão pouco a sua interface de programação.
- Membros privados não podem ser utilizados externamente as classes que os declaram

Herança

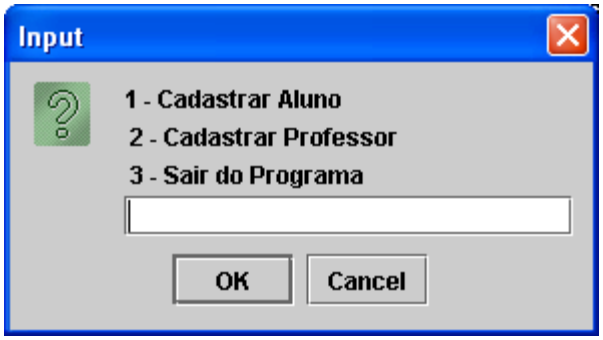
- O acesso protegido (*protected*) e público (*public*) servem, respectivamente, para definir membros destinado a interface de programação e a interface de usuário.
- Os membros públicos podem, sem restrição podem ser utilizados por instâncias da classe, por subclasses e por instâncias das subclasses.
- Membros protegidos podem ser utilizados apenas na implementação de subclasses, não sendo acessíveis por instâncias da classe ou por instâncias das subclasses.

```
// classe Pessoa
import javax.swing.*;
public class Pessoa
{
   private
     String nome; //nome da pessoa
     String endereco; // endereco da pessoa
     String cpf; // cpf da pessoa

   public Pessoa( ) //contrutor default da classe
     {
        setPessoa(" "," "," ");
     }

   //métodos set

   public void setPessoa(String n, String e, String c)
   {
      nome = n;
      endereco = e;
      cpf = c;
   }
   public void setNome(String n)  //configurar o nome
   {
      nome = n;
   }

   public void setEndereco(String e) //configurar o endereco
   {
      endereco = e;
   }
   public void setCpf(String c)  //configurar o cpf
     {
        cpf = c;
     }

  //métodos get
     public String getNome( )
     {    return nome;  }

     public String getEndereco( )
     {      return endereco;   }

     public String getCpf( )
     {  return cpf;  }

     public void Cadastrar_Pessoa()
     {
        nome =JOptionPane.showInputDialog("Informe
                o Nome");
        endereco = JOptionPane.showInputDialog(
                "Informe o Endereco");
        cpf = JOptionPane.showInputDialog("Informe o
                CPF");
     }

  } //fim da classe
```

```
import javax.swing.JOptionPane;

// classe Pessoa

public class Professor extends Pessoa
{
   private
     double salario; //salário do professor
     String titulacao; // grau de instrução do professor

   public Professor( ) //contrutor default da classe
     {
        super ( );
        salario = 0;
        titulacao = "";
     }

   //métodos set

   public void setProfessor(String n, String e, String c,
                                    double s, String t)
   {
      setPessoa(n, e, c);
      salario = s;
      titulacao = t;
   }
   public void setSalario(double s)  //configurar o salario
   {   salario = s;  }

   public void setTitulacao(String t) //configurar a titulacao
   {   titulacao = t;   }

  //métodos get
    public double getSalario( )
    {
       return salario;
    }

    public String getTitulacao( )
    {
       return titulacao;
    }
    public void Cadastrar_Professor()
    {
         Cadastrar_Pessoa( );
         salario = Double.parseDouble(
                JOptionPane.showInputDialog(
                          "Informe o Salário"));
         titulacao = JOptionPane.showInputDialog(
                          "Informe a Titulação");

    }
}
```

```
import javax.swing.JOptionPane;

// classe Pessoa

public class Aluno extends Pessoa
{
   private
     int anoingresso; //ano de ingresso do aluno
     String curso; // curso matriculado do aluno

   public Aluno( ) //contrutor default da classe
     {
        super ();
        anoingresso = 0;
        curso = "";
     }

   //métodos set

   public void setProfessor(String n, String e, String c, int a,
                                                  String cur)
   {
      setPessoa(n, e, c);
      anoingresso = a;
      curso = cur;
   }
   public void setAnoIngresso(int a)  //configurar o salario
   {   anoingresso = a;  }

   public void setCurso(String cur) //configurar a titulação
   {   curso = cur;   }

  //métodos get
    public int getAnoIngresso( )
    {
       return anoingresso;
    }

    public String getCurso( )
    {
       return curso;
    }
    public void Cadastrar_Aluno()
    {
          Cadastrar_Pessoa( );
          anoingresso = Integer.parseInt(
                JOptionPane.showInputDialog(“
                Informe o ano de ingresso"));
          curso = JOptionPane.showInputDialog(
                          "Informe o curso");
    }

}
```

```
import javax.swing.*;

public class Faculdade
{
   public static void main( String args[] )
   {
      int opc;
      Professor p = new Professor();
      Aluno a = new Aluno();
      do
      {
        opc = Integer.parseInt(
              JOptionPane.showInputDialog("1 - Cadastrar Aluno \n 2 - Cadastrar Professor \n 3 - Sair
                 do Programa"));
        switch (opc)
        {
           case 1:  a.Cadastrar_Aluno();
           break;
           case 2:  p.Cadastrar_Professor();
           break;
           case 3:  break;
           default: JOptionPane.showMessageDialog(null, "Opção Inválida.");
        }
      }while (opc != 3);
      System.exit(0);
   }
}
```



Polimorfismo

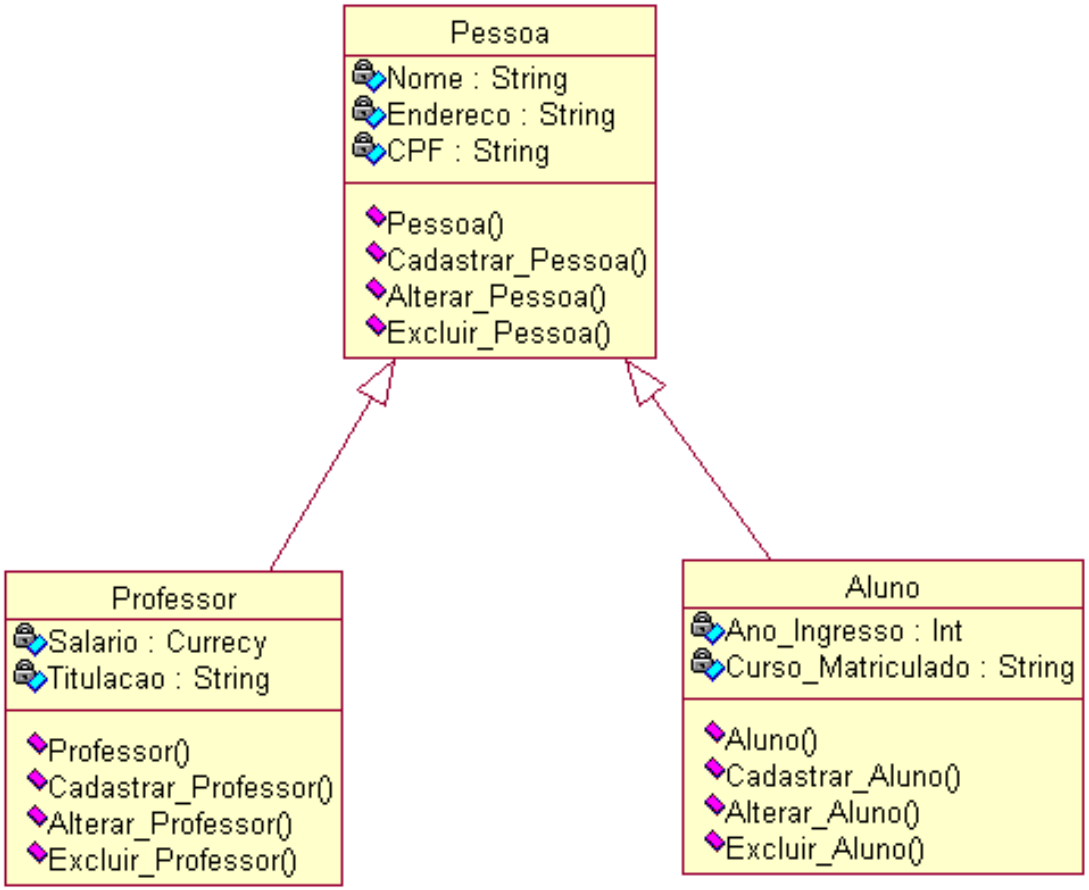
- Através dos mecanismos de encapsulamento, novos métodos e atributos podem ser adicionados sem alteração das interfaces de programação e de usuário existentes de uma certa classe, constituindo um poderoso mecanismo de extensão de classes, mas o mecanismo de herança oferece possibilidades muito maiores.
- A herança permite:
  - que novas classes sejam criadas e adicionadas a uma hierarquia sem a necessidade de qualquer modificação do código existente das classes e das aplicações que utilizam estas classes pois cada classe define estritamente seus próprios métodos e atributos.
  - As novas classes adicionadas a hierarquia podem estender, especializar ou mesmo restringir o comportamento das classes originais.

Polimorfismo

- Mas quando analisamos a hierarquia das classes no sentido inverso, isto é, nos dirigindo a superclasse da hierarquia, ocorre a generalização das diferentes classes interligadas. Isto significa que todas as classes pertencentes a uma mesma família de classes podem ser genericamente tratadas como sendo uma classe do tipo mais primitivo existente, ou seja, como sendo da própria superclasse da hierarquia.

Polimorfismo

- Na figura abaixo temos:



Polimorfismo

- Podemos perceber que a classe **Pessoa**, como superclasse desta hierarquia, oferece tratamento genérico para as classe **Professor** e **Aluno** bem como para quaisquer novas classes que sejam derivadas destas últimas ou da própria superclasse.
- O tratamento generalizado de classes permite escrever programas que podem ser mais facilmente extensíveis, isto é, que podem acompanhar a evolução de uma hierarquia de classe sem necessariamente serem modificados.
- Enquanto a herança oferece um poderoso mecanismo de especialização, o polimorfismo oferece em igualmente poderoso mecanismo de generalização, constituindo uma outra dimensão da separação da interface de uma classe de sua efetiva implementação.

Polimorfismo

- Imaginemos que seja necessário implementar numa classe **Faculdade** um método que seja capaz de imprimir o nome de objetos Pessoas os quais podem ser tanto do tipo:
  - Aluno como Professor.
- Poderíamos, utilizando a sobrecarga, criar um método de mesmo nome que recebesse ou um Aluno ou um Professor:

```
public class Faculdade
{
    public void Nome (Aluno a)
    {
        System.out.println(“Nome: "+ a.getNome ());
    }

    public void Nome (Professor p)
    {
        System.out.println(“Nome: "+p.getNome ());
    }
}
```

Polimorfismo

- Embora correto, existem duas grandes desvantagens nesta aproximação:
  - (i) para cada tipo de Pessoa existente teríamos um método sobrecarregado avolumando o código que deveria ser escrito (e depois mantido) além disso
  - (ii) a adição de novas subclasse que representassem outros tipos de Pessoas exigiria a implementação adicional de outros métodos semelhantes.

- Através do polimorfismo pode-se chegar ao mesmo resultado mas de forma mais simples pois a classe Pessoa permite o tratamento generalizado de todas as suas subclasses.

Polimorfismo

```
public class Faculdade
{
	public void Nome (Pessoa pe)
	{
		System.out.println(“Nome: "+pe.getNome ());
	}
}
```

- Nesta situação o polimorfismo permite que um simples trecho de código seja utilizado para tratar objetos diferentes relacionados através de seu ancestral comum:
  - simplificando a implementação,
  - melhorando a legibilidade do programa
  - e também aumentando sua flexibilidade.

Polimorfismo

- Outra situação possível é a seguinte: como qualquer método de uma superclasse pode ser sobreposto (*overhided*) em uma subclasse, temos que o comportamento deste método pode ser diferente em cada uma das subclasses, no entanto através da superclasse podemos genericamente usar tal método, que produz um resultado distinto para cada classe utilizada.
- Assim o polimorfismo também significa que uma mesma operação pode se comportar de forma diferente em classes diferentes.
- O polimorfismo é uma característica importantíssima oferecida pela orientação à objetos que se coloca ao lado dos mecanismos de abstração de dados, herança e encapsulamento.

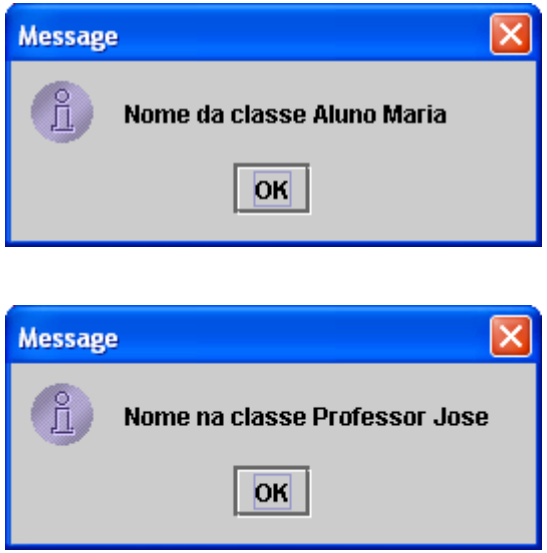
```
//definição da classe pessoa
public class Pessoa
{
   public
     String nome;
   public Pessoa()
   {}

   public abstract String getNome();
}
```

```
public class Aluno extends Pessoa
{
    public Aluno()
    {
       nome = "Maria";
    }
       public String getNome()
   {
          return "Nome da classe Aluno " + nome;
   }
}
```

```
public class Professor extends Pessoa
{
       public Professor()
       {
                nome = "Jose";
       }
          public String getNome()
          {
                   return "Nome na classe Professor " + nome;
          }
}
```

Polimorfismo

```
import javax.swing.*;
public class TestPessoa {

    public static void main(String[] args)
    {
        Aluno a = new Aluno();
        Professor p = new Professor();
        Imprime(a);
        Imprime(p);


    }
    public static void Imprime(Pessoa pe)
    {
        JOptionPane.showMessageDialog(null, pe.getNome());
    }

}
```



Métodos e classes final

- Variáveis declaradas final indicam que não podem ser modificadas depois que são declaradas e que devem ser inicializadas quando declaradas.
- Também é possível definir métodos e classes com o modificador final
- Um método declarado final não pode ser redefinido em uma subclasse.
- Uma classe que é declarada final não pode ser uma superclasse (isto é, uma classe não pode herdar de uma classe final).
- Todos os métodos em uma classe final são implicitamente final.

Superclasses abstratas

- Em algumas circunstâncias desejamos orientar como uma classe deve ser implementada, ou melhor, como deve ser a interface de uma certa classe.
- Em outros casos, o modelo representado é tão amplo que certas classes tornam-se por demais gerais, não sendo possível ou razoável que possuam instâncias. Para estes casos dispomos das classes abstratas (*abtract classes*).
- Tais classes são assim denominadas por não permitirem a instanciação, isto é, por não permitirem a criação de objetos do seu tipo.
- Sendo assim seu uso é dirigido para a construção de classes modelo, ou seja, de especificações básicas de classes através do mecanismo de herança.

Superclasses abstratas

- Uma classe abstrata deve ser estendida, ou seja, deve ser a classe base de outra, mais específica que contenha os detalhes que não puderam ser incluídos na superclasse (abstrata).
- Outra possível aplicação das classes abstratas é a criação de um ancestral comum para um conjunto de classes que, se originados desta classe abstrata, poderão ser tratados genericamente através do polimorfismo.
- Um classe abstrata, como qualquer outra, pode conter métodos mas também podem adicionalmente conter métodos abstratos, isto é, métodos que deverão ser implementados em suas subclasses não abstratas.