Faculdade de Computação - FACOM

Bacharelado em Sistemas de Informação

FACOM32305 - Programação Orientada a Objetos

Prof. Thiago Pirola Ribeiro

Sumário

Modificador static para métodos

- A maioria dos métodos é executada em resposta a chamadas de método em objetos específicos.

Modificador static para métodos

- A maioria dos métodos é executada em resposta a chamadas de método em objetos específicos.
- Às vezes um método realiza uma tarefa que não depende do conteúdo de nenhum objeto.

Modificador static para métodos

- A maioria dos métodos é executada em resposta a chamadas de método em objetos específicos.
- Às vezes um método realiza uma tarefa que não depende do conteúdo de nenhum objeto.
- Esse método se aplica à classe em que é declarado como um todo, e é conhecido como método static ou método de classe.

Modificador static para métodos

- A maioria dos métodos é executada em resposta a chamadas de método em objetos específicos.
- Às vezes um método realiza uma tarefa que não depende do conteúdo de nenhum objeto.
- Esse método se aplica à classe em que é declarado como um todo, e é conhecido como método static ou método de classe.
- Para declarar um método como static, coloque a palavra-chave static antes do tipo de retorno na declaração do método.

Modificador static para métodos

- A maioria dos métodos é executada em resposta a chamadas de método em objetos específicos.
- Às vezes um método realiza uma tarefa que não depende do conteúdo de nenhum objeto.
- Esse método se aplica à classe em que é declarado como um todo, e é conhecido como método static ou método de classe.
- Para declarar um método como static, coloque a palavra-chave static antes do tipo de retorno na declaração do método.
- Pode-se chamar um método static especificando o nome da classe na qual este é declarado, seguido por um ponto e pelo nome do método (Ex.: *Math.sqrt(argumentos)*).

Modificador static para métodos

- A maioria dos métodos é executada em resposta a chamadas de método em objetos específicos.
- Às vezes um método realiza uma tarefa que não depende do conteúdo de nenhum objeto.
- Esse método se aplica à classe em que é declarado como um todo, e é conhecido como método static ou método de classe.
- Para declarar um método como static, coloque a palavra-chave static antes do tipo de retorno na declaração do método.
- Pode-se chamar um método static especificando o nome da classe na qual este é declarado, seguido por um ponto e pelo nome do método (Ex.: *Math.sqrt(argumentos)*).
- Não é necessário que algum objeto tenha sido criado para que o método deste tipo possa ser chamado.

Modificador static para atributos

- Cada objeto tem sua própria cópia de todas as variáveis de instância da classe.

Modificador static para atributos

- Cada objeto tem sua própria cópia de todas as variáveis de instância da classe.
- Quando apenas uma cópia de uma variável particular precisa ser compartilhada por todos os objetos de uma classe, utilizamos um campo static - chamado de variável de classe.

Modificador static para atributos

- Cada objeto tem sua própria cópia de todas as variáveis de instância da classe.
- Quando apenas uma cópia de uma variável particular precisa ser compartilhada por todos os objetos de uma classe, utilizamos um campo static - chamado de variável de classe.
- Este tipo de variável representa informações de escopo de classe.

Modificador static para atributos

- Cada objeto tem sua própria cópia de todas as variáveis de instância da classe.
- Quando apenas uma cópia de uma variável particular precisa ser compartilhada por todos os objetos de uma classe, utilizamos um campo static - chamado de variável de classe.
- Este tipo de variável representa informações de escopo de classe.
- Estas podem ser acessadas mesmo se nenhum objeto da classe tiver sido criado.

Modificador static para atributos

- Cada objeto tem sua própria cópia de todas as variáveis de instância da classe.
- Quando apenas uma cópia de uma variável particular precisa ser compartilhada por todos os objetos de uma classe, utilizamos um campo static - chamado de variável de classe.
- Este tipo de variável representa informações de escopo de classe.
- Estas podem ser acessadas mesmo se nenhum objeto da classe tiver sido criado.
- Para declarar um atributo como static, coloque a palavra-chave static antes do tipo e nome do atributo.

Modificador static para atributos

- Pode-se acessar um atributo public static especificando o nome da classe na qual este é declarado, seguido por um ponto e pelo nome do atributo (Ex.: *Math.PI*).

Modificador static para atributos

- Pode-se acessar um atributo public static especificando o nome da classe na qual este é declarado, seguido por um ponto e pelo nome do atributo (Ex.: *Math.PI*).
- Se o atributo for privado, é necessário criar um método public static para retornar o valor deste atributo.

Modificador static para atributos

- Pode-se acessar um atributo public static especificando o nome da classe na qual este é declarado, seguido por um ponto e pelo nome do atributo (Ex.: *Math.PI*).
- Se o atributo for privado, é necessário criar um método public static para retornar o valor deste atributo.
- Métodos deste tipo não podem acessar membros de classe não static, pois métodos static podem ser chamados mesmo quando nenhum objeto da classe foi instanciado.

Modificador final para variáveis

- Este modificador é utilizado para especificar o fato de que uma variável não é modificável (é uma constante) e que qualquer tentativa de modificá-la é um erro.

Modificador final para variáveis

- Este modificador é utilizado para especificar o fato de que uma variável não é modificável (é uma constante) e que qualquer tentativa de modificá-la é um erro.
- Elas podem ser inicializadas quando declaradas, ou por cada um dos construtores da classe no caso de cada objeto ter um valor diferente.

Modificador final para variáveis

- Este modificador é utilizado para especificar o fato de que uma variável não é modificável (é uma constante) e que qualquer tentativa de modificá-la é um erro.
- Elas podem ser inicializadas quando declaradas, ou por cada um dos construtores da classe no caso de cada objeto ter um valor diferente.
- Tentar modificar uma variável de instância final depois que ela é inicializada é um erro de compilação.

Modificador final para variáveis

- Este modificador é utilizado para especificar o fato de que uma variável não é modificável (é uma constante) e que qualquer tentativa de modificá-la é um erro.
- Elas podem ser inicializadas quando declaradas, ou por cada um dos construtores da classe no caso de cada objeto ter um valor diferente.
- Tentar modificar uma variável de instância final depois que ela é inicializada é um erro de compilação.
- Se a variável final não for inicializada, ocorrerá erro de compilação.

Modificador final para variáveis

- Este modificador é utilizado para especificar o fato de que uma variável não é modificável (é uma constante) e que qualquer tentativa de modificá-la é um erro.
- Elas podem ser inicializadas quando declaradas, ou por cada um dos construtores da classe no caso de cada objeto ter um valor diferente.
- Tentar modificar uma variável de instância final depois que ela é inicializada é um erro de compilação.
- Se a variável final não for inicializada, ocorrerá erro de compilação.
- Deve-se usar um campo final e static se ele for inicializado em sua declaração com um valor que é o mesmo de todos os objetos da classe, e não irá mudar.

Modificador final para variáveis

- Este modificador é utilizado para especificar o fato de que uma variável não é modificável (é uma constante) e que qualquer tentativa de modificá-la é um erro.
- Elas podem ser inicializadas quando declaradas, ou por cada um dos construtores da classe no caso de cada objeto ter um valor diferente.
- Tentar modificar uma variável de instância final depois que ela é inicializada é um erro de compilação.
- Se a variável final não for inicializada, ocorrerá erro de compilação.
- Deve-se usar um campo final e static se ele for inicializado em sua declaração com um valor que é o mesmo de todos os objetos da classe, e não irá mudar.
- Para declarar, basta acrescentar a palavra final antes do tipo e nome do atributo.

Modificador final para métodos e classes

- Um método final em uma superclasse não pode ser sobrescrito em uma subclasse.

Modificador final para métodos e classes

- Um método final em uma superclasse não pode ser sobrescrito em uma subclasse.
- Uma classe final que é declarada final não pode ser uma superclasse (isto é, uma classe não pode estender uma classe final).

Modificador final para métodos e classes

- Um método final em uma superclasse não pode ser sobrescrito em uma subclasse.
- Uma classe final que é declarada final não pode ser uma superclasse (isto é, uma classe não pode estender uma classe final).
- Todos os métodos em uma classe final são implicitamente final.

Sumário

Polimorfismo

Termo originário do grego *poly* (muitas) + *morpho* (formas).

O polimorfismo em POO é a habilidade de objetos de uma ou mais classes em responder a uma mesma mensagem de *formas diferentes*.

Polimorfismo

- Métodos com mesmo nome, mas implementados de maneira diferente.

Polimorfismo

- Métodos com mesmo nome, mas implementados de maneira diferente.
- Permite obter códigos genéricos:

Polimorfismo

- Métodos com mesmo nome, mas implementados de maneira diferente.
- Permite obter códigos genéricos:
  - Processar diversos tipos de dados;

Polimorfismo

- Métodos com mesmo nome, mas implementados de maneira diferente.
- Permite obter códigos genéricos:
  - Processar diversos tipos de dados;
  - Processar os dados de formas distintas;

Polimorfismo

- Métodos com mesmo nome, mas implementados de maneira diferente.
- Permite obter códigos genéricos:
  - Processar diversos tipos de dados;
  - Processar os dados de formas distintas;
  - Podem fazer um mesmo objeto ter comportamentos diferentes para uma mesma ação/solicitação.

Polimorfismo

Veremos a ocorrência do polimorfismo de duas maneiras:

- Sobrecarga (*Overloading*)

Polimorfismo

Veremos a ocorrência do polimorfismo de duas maneiras:

- Sobrecarga (*Overloading*)
- Sobreposição ou sobreescrita (*Overriding*)

Polimorfismo

Veremos a ocorrência do polimorfismo de duas maneiras:

- Sobrecarga (*Overloading*)
- Sobreposição ou sobreescrita (*Overriding*)

Alguns autores não classificam a sobrecarga como um tipo de polimorfismo.

Sobrecarga

Permite que um método seja definido com diferentes *assinaturas* e diferentes implementações.

Sobrecarga

Permite que um método seja definido com diferentes *assinaturas* e diferentes implementações.

- Assinatura: relacionada ao *número* e *tipo* dos parâmetros.

Sobrecarga

Permite que um método seja definido com diferentes *assinaturas* e diferentes implementações.

- Assinatura: relacionada ao *número* e *tipo* dos parâmetros.

Resolvido pelo compilador em tempo de compilação:

Sobrecarga

Permite que um método seja definido com diferentes *assinaturas* e diferentes implementações.

- Assinatura: relacionada ao *número* e *tipo* dos parâmetros.

Resolvido pelo compilador em tempo de compilação:

- A assinatura diferente permite ao compilador dizer qual dos *sinônimos* será utilizado.

Sobrecarga

Permite que um método seja definido com diferentes *assinaturas* e diferentes implementações.

- Assinatura: relacionada ao *número* e *tipo* dos parâmetros.

Resolvido pelo compilador em tempo de compilação:

- A assinatura diferente permite ao compilador dizer qual dos *sinônimos* será utilizado.

Exemplo: quando definimos diferentes construtores em uma classe, cada um recebendo parâmetros diferentes.

Sobrecarga

Atenção

Mudar o nome dos parâmetros não é uma sobrecarga, o compilador diferencia o tipo, e não o nome dos parâmetros.

Sobrecarga

Atenção

Mudar o nome dos parâmetros não é uma sobrecarga, o compilador diferencia o tipo, e não o nome dos parâmetros.

Exemplo: métodos

```
f(int a, int b) e
f(int c, int d)
```

em uma mesma classe resultam em *erro de redeclaração*.

Sobrecarga

Como dito, as assinaturas devem ser diferentes.

Sobrecarga

Como dito, as assinaturas devem ser diferentes. O que é a assinatura?

Sobrecarga

Como dito, as assinaturas devem ser diferentes. O que é a assinatura?

A assinatura de um método é composta pelo nome do método e pelos tipos dos seus argumentos, *independente dos nomes dos argumentos e do valor de retorno da função*.

Sobrecarga

Como dito, as assinaturas devem ser diferentes. O que é a assinatura?

A assinatura de um método é composta pelo nome do método e pelos tipos dos seus argumentos, *independente dos nomes dos argumentos e do valor de retorno da função*.

Ex.: 2 assinaturas iguais:

```
float soma(float a, float b);
void soma(float op1, float op2);
```

Sobrecarga

Como dito, as assinaturas devem ser diferentes. O que é a assinatura?

A assinatura de um método é composta pelo nome do método e pelos tipos dos seus argumentos, *independente dos nomes dos argumentos e do valor de retorno da função*.

Ex.: 2 assinaturas iguais:

```
float soma(float a, float b);
void soma(float op1, float op2);
```

Ex.: 2 assinaturas diferentes:

```
float soma(float a, float b);
double soma(double a, double b);
```

Sobrecarga

É implementada, normalmente, para métodos que devem executar operações semelhantes, usando uma lógica de programação diferente para diferentes tipos de dados.

Sobrecarga

É implementada, normalmente, para métodos que devem executar operações semelhantes, usando uma lógica de programação diferente para diferentes tipos de dados.

```
1  public class Funcoes{
2      public static int quadrado( int x ) {
3          return x * x;
4      }
5
6      public static double quadrado( double y ) {
7          return y * y;
8      }
9  }
10 // ...
11 System.out.println("2 ao quadrado: " + Funcoes.quadrado(2));
12 System.out.println("PI ao quadrado: " + Funcoes.quadrado(Math.PI));
```

Sobrecarga

Em muitos casos é necessário criar métodos que precisam de mais ou menos parâmetros, ou até precisem de parâmetros de tipos diferentes.

Sobrecarga

Em muitos casos é necessário criar métodos que precisam de mais ou menos parâmetros, ou até precisem de parâmetros de tipos diferentes.

```
1 Fracao f1 = new Fracao();  // num=0,den=1
2 Fracao f2 = new Fracao(f1); // copy constructor: copia conteúdo de
      f1
3 Fracao f3 = new Fracao(5);  // num=5, den=1 (inteiro)
4 Fracao f4 = new Fracao(5,2);
```

Sobrecarga

```
1  public class Fracao {
2      private int num, den;
3
4      public Fracao(){
5          den=1;
6          num=0;
7      }
8      public Fracao(int a){
9          num=a;
10         den=1;
11     }
12     public Fracao(int a, int b){
13         num=a;
14         den=b;
15     }
16     public Fracao(Fracao A){
17         num=A.num;
18         den = A.den;
19     }
20     public void mostrar(){
21         System.out.printf("\n%d/%d", num, den);
22     }
23 }
24
```

Sobrecarga

```
1  public static void main(String[] args) {
2
3      Fracao f1 = new Fracao();
4      f1.mostrar();
5
6      Fracao f2 = new Fracao(f1);
7      f2.mostrar();
8
9      Fracao f3 = new Fracao(4);
10     f3.mostrar();
11
12     Fracao f4 = new Fracao(7,2);
13     f4.mostrar();
14 }
15
```

Sobrecarga

```
1  public static void main(String[] args) {
2
3      Fracao f1 = new Fracao();
4      f1.mostrar();  // 0/1
5
6      Fracao f2 = new Fracao(f1);
7      f2.mostrar();   // 0/1
8
9      Fracao f3 = new Fracao(4);
10     f3.mostrar();    // 4/1
11
12     Fracao f4 = new Fracao(7,2);
13     f4.mostrar();    // 7/2
14 }
15
```

Sobrescrita

Conceito já visto em herança:

Sobrescrita

Conceito já visto em herança:

- Permite a redefinição do funcionamento de um método herdado de uma *classe base*.

Sobrescrita

Conceito já visto em herança:

- Permite a redefinição do funcionamento de um método herdado de uma *classe base*.
- A *classe derivada* tem uma função com a mesma assinatura da classe base, mas funcionamento diferente;

Sobrescrita

Conceito já visto em herança:

- Permite a redefinição do funcionamento de um método herdado de uma *classe base*.
- A *classe derivada* tem uma função com a mesma assinatura da classe base, mas funcionamento diferente;
- O método na classe derivada sobrepõe a função na classe base.

Sobrescrita

Polimorfismo estático × Polimorfismo dinâmico

Sobrescrita: polimorfismo dinâmico – envolve 2 classes (classe derivada herda e redefine método da classe base);

Sobrescrita

Polimorfismo estático × Polimorfismo dinâmico

Sobrescrita: polimorfismo dinâmico – envolve 2 classes (classe derivada herda e redefine método da classe base);

Sobrecarga: Polimorfismo estático – métodos com mesmo nome e assinaturas diferentes na mesma classe.

Sobrescrita

Indicações para uso da sobrescrita:

Sobrescrita

Indicações para uso da sobrescrita:

- A implementação do método na classe base não é adequada na classe derivada;

Sobrescrita

Indicações para uso da sobrescrita:

- A implementação do método na classe base não é adequada na classe derivada;
- A classe base não oferece implementação para o método, somente a declaração;

Sobrescrita

Indicações para uso da sobrescrita:

- A implementação do método na classe base não é adequada na classe derivada;
- A classe base não oferece implementação para o método, somente a declaração;
- A classe derivada pretende estender as funcionalidades da classe base.

Sobrescrita

Exemplo: considere as seguintes classes:

```
public abstract class Animal {
    protected double peso;
    protected int idade;
    protected int membros;

    public abstract void locomover();
    public abstract void alimentar();
    public abstract void emitirSom();

    // getters e setters
}
```

Sobrescrita

```
public class Mamifero extends Animal {
    private String corPelo;

    //@Override
    public void locomover() {
        System.out.println("Correndo.");
    }

    //@Override
    public void alimentar() {
        System.out.println("Mamando.");

    }

    //@Override
    public void emitirSom() {
        System.out.println("Som de mamífero.");
    }

    /// get e set corPelo
}
```

Sobrescrita

```
public class Cachorro extends Mamifero {

    //@Override
    public void emitirSom() {
        System.out.println("Au au au.");
    }

    public void enterrarOsso() {
        System.out.println("Enterrando osso.");
    }
    public void abanarRabo() {
        System.out.println("Abanando o rabo.");
    }
}
```

Sobrescrita

```
public class ExemploPoli {
    public static void main(String[] args) {
        System.out.println("----MAMÍFERO----");
        Mamifero mamifero = new Mamifero();
        mamifero.setCorPelo("Preto");
        mamifero.setPeso(86.7);
        mamifero.setIdade(42);
        mamifero.alimentar();
        System.out.println("Pelo:"+mamifero.getCorPelo());
        System.out.println("Peso:"+mamifero.getPeso());

        System.out.println("\n----CACHORRO----");
        Mamifero cachorro = new Cachorro();
        cachorro.setPeso(7.23);
        cachorro.setIdade(4);
        cachorro.alimentar();
        cachorro.locomover();
        System.out.println("Peso:"+cachorro.getPeso());
    }
}
```

Sobrescrita

Resultado:

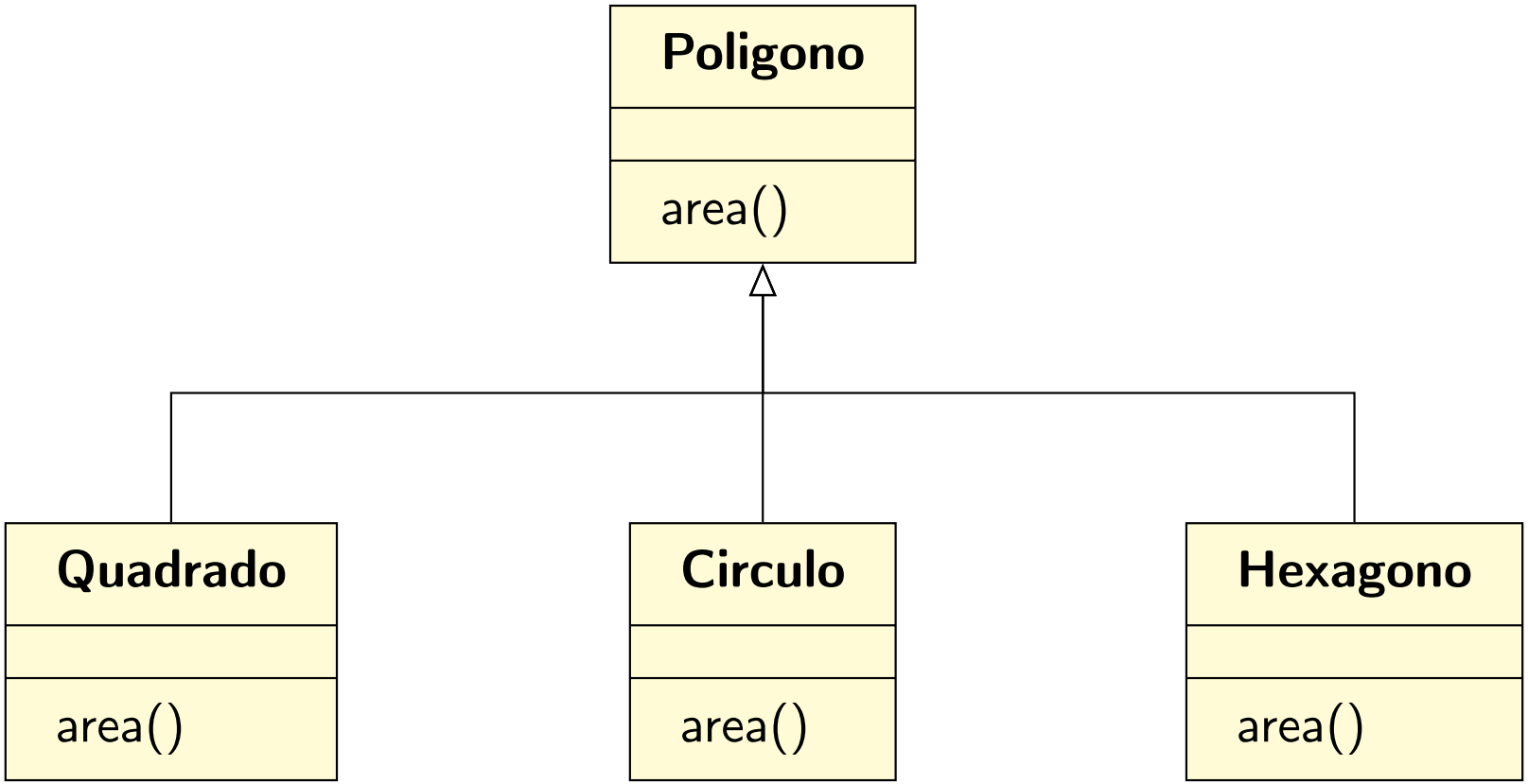
```
----MAMÍFERO----
Mamando.
Pelo:Preto
Peso:86.7

----CACHORRO----
Mamando.
Correndo.
Peso:7.23
```

Na ocasião desta chamada, será decidido *automaticamente* qual implementação será invocada, dependendo do objeto: esta decisão é denominada ligação dinâmica (*dynamic binding*).

Exemplo de polimorfismo I

Considere o polimorfismo a seguir, *em métodos*:



Exemplo de polimorfismo II

Seguindo o exemplo, podemos observar o polimorfismo *nas variáveis*:

- Uma variável do tipo `Poligono` pode assumir a forma de `Poligono`, `Quadrado`, `Círculo`, etc.

```
1 static Poligono pol[] = new Poligono[2];
2 static int MAXPOLIG = 2;
3
4 Poligono p1 = new Quadrado(10);
5 Poligono p2 = new Circulo(10);
6
7 pol[0] = p1;
8 pol[1] = p2;
9 ...
```

Exemplo sem polimorfismo I

Área total de um *array* de polígonos, usando exemplo anterior, mas sem sobrescrita de `area()`:

```
1  double areaTotal() {
2      double areaTotal = 0;
3      for (int i = 0; i < MAXPOLIG; ++i) {
4          if (pol[i] instanceof Poligono)
5              areaTotal +=
6              pol[i].areaPoligono();
7          else if (pol[i] instanceof Triangulo)
8              areaTotal +=
9              pol[i].areaTriangulo();
10         else if (pol[i] instanceof Retangulo)
11             areaTotal +=
12             pol[i].areaRetangulo();
```

Exemplo sem polimorfismo II

```
13          else if (pol[i] instanceof Hexagono)
14              areaTotal +=
15              pol[i].areaHexagono();
16          return areaTotal;
17      }
18  }
```

`instanceof`: palavra reservada para testar se objeto é de determinada classe, retornando `true` quando for o caso, e `false` caso contrário.

Exemplo com polimorfismo

Usamos polimorfismo de `area()` como no diagrama de classes visto:

```
1 double areaTotal() {
2     double areaTotal = 0;
3     for (int i = 0; i < MAXPOLIG; ++i) {
4         areaTotal += pol[i].area();
5     return areaTotal;
6 }
```

Rápido, enxuto e fácil de entender:

- O acréscimo de uma nova subclasse de `Poligono` não altera nenhuma linha do código acima.

```java
public abstract class Poligono {
      public abstract double area();
}

public class Quadrado extends Poligono {
   private double lado;
   public Quadrado(double lado) {
      this.lado = lado;
   }
   //@Override
   public double area() {
      return lado * lado;
   }
}

public class Circulo extends Poligono {
   private double raio;
   public Circulo(double raio) {
      this.raio = raio;
   }
   //@Override
   public double area() {
      return Math.PI * (raio * raio);
   }
}
```

```
public class TestaClasses{

   static Poligono pol[] = new Poligono[2];
   static int MAXPOLIG = 2;
   public static void main(String[] args) {

      Poligono p1 = new Quadrado(10);
      Poligono p2 = new Circulo(10);

      pol[0] = p1;
      pol[1] = p2;

      System.out.println("\n\nÁrea Total: "+areaTotal());
   }

   public static double areaTotal() {
      double areaTotal = 0;
      for (int i = 0; i < MAXPOLIG; ++i){
         System.out.println("\nÁrea do " +
            pol[i].getClass().getName() + ": " + pol[i].area());
         areaTotal += pol[i].area();
      }
      return areaTotal;
   }
}
```

Benefícios do polimorfismo

Legibilidade do código:

- O mesmo nome para a mesma operação (método) facilita o aprendizado e melhora a legibilidade.

Código de menor tamanho:

- Código mais claro, enxuto e elegante.

Flexibilidade:

- Pode-se incluir novas classes sem alterar o código que a manipulará.

Casting

- Uma referência de superclasse só pode ser utilizada para invocar os métodos declarados na superclasse.

```
1 //Pessoa é superclasse e Aluno é subclasse.
2 Pessoa P = new Aluno();
```

Casting

- Uma referência de superclasse só pode ser utilizada para invocar os métodos declarados na superclasse.

```
1 //Pessoa é superclasse e Aluno é subclasse.
2 Pessoa P = new Aluno();
```

```
1 // ERRO: Não é possível acessar o método que
  existe apenas na subclasse Aluno, e não na
  superclasse Pessoa.
2 P.matricularAluno();
```

Casting

- O compilador Java permite a atribuição de uma referência de superclasse a uma variável de subclasse se fizermos explicitamente uma coerção (casting) da referência de superclasse para o tipo de subclasse.

```
1  //Pessoa é superclasse e Aluno é subclasse.
2  Pessoa P = new Aluno();
3  if (P instanceof Aluno){
4    // Casting para acessar especificidades da
   classe Aluno.
5      Aluno A = (Aluno) P;
6      // Agora o método pode ser acessado.
7      A.matricularAluno();
8  }
```

Referências

1. Addison-BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML, Guia do Usuário*. Rio de Janeiro: Campus, 2000.
2. FOWLER, M. *UML Essencial*, 2a Edição. Porto Alegre: Bookman, 2000.
3. LARMAN, C. *Utilizando UML e Padrões: Uma Introdução à Análise e ao Projeto Orientado a Objetos*. Porto Alegre: Bookman, 2001.
4. Paul Deitel e Harvey Deitel, Java: como programar, Editora Pearson, 8a edição, 2010.

Grande parte dos slides dessa apresentação foram cedidos por:

- Graça Marietto e Francisco Zampirolli, UFABC
- Profa Katti Faceli, UFSCar/Sorocaba
- Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU