Linguagem de Programação I

Aula 6
Programação Orientada a Objetos –
Polimorfismo, Atributos e Métodos Finais e Estáticos, Classes Abstratas e Interfaces

l.bertholdo@ifsp.edu.br

Conteúdo

- Polimorfismo
  - Sobrecarga de Métodos (*Overload*)
  - Sobreposição de Métodos (*Override*)
- Atributos, Métodos e Classes Finais
- Atributos e Métodos Estáticos
- Classes Abstratas
- Interfaces

l.bertholdo@ifsp.edu.br

Polimorfismo

*O termo **polimorfismo** origina-se do grego (**poli:** muitas, **morphos:** formas).*

- Recurso que permite definir **comportamentos diferentes** para métodos que possuem:
  - O **mesmo nome** e se encontram na mesma classe.
  - A **mesma assinatura** e existem nas classes mãe e filhas.
- Ao usar polimorfismo, evita-se criar métodos com nomes diferentes para realizar funções que, conceitualmente, são as mesmas.
- Existem dois tipos de polimorfismo:
  - Sobrecarga de Métodos (*Overload*)
  - Sobreposição (ou Sobrescrição) de Métodos (*Overload*)

Assinatura de método: *Tipo de retorno + Nome + Argumentos e Tipos de argumentos*
Exemplo: *double calculaAreaTriangulo(double base, double altura)*

l.bertholdo@ifsp.edu.br

Sobrecarga de Métodos (*Overload*)

- Tipo de polimorfismo que não envolve herança, também conhecido como **polimorfismo estático**.
- Ocorre quando a classe possui dois ou mais métodos com o **mesmo nome**, porém com **assinaturas diferentes**, ou seja, quantidades de argumentos e/ou tipos de argumentos diferentes.
- A decisão de qual método chamar é tomada em **tempo de compilação**, baseada nos argumentos que foram passados na chamada do método.
- A sobrecarga é muito usada em **construtores** e pode ser muito útil quando se deseja inicializar os atributos de uma classe de diferentes formas.

Sobrecarga de Métodos

```java
public class Inicio {
    public static void main(String[] args) {
        Calculo c = new Calculo();
        c.somar(8, 0, 2); // Chama o 3º método "somar" da classe Calculo.
        c.somar(3, 4);    // Chama o 1º método "somar" da classe Calculo.
        c.somar(6, 2.3);  // Chama o 2º método "somar" da classe Calculo.
    }
}
```

```java
public class Calculo {
    public void somar(int n1, int n2){
        System.out.println("Soma: " + (n1 + n2));
    }

    public void somar(int n1, double n2) {
        System.out.println("Soma: " + (n1 + n2));
    }

    public void somar(int n1, int n2, int n3) {
        System.out.println("Soma: " + (n1 + n2 + n3));
    }
}
```

```
Soma: 10
Soma: 7
Soma: 8.3
```

l.bertholdo@ifsp.edu.br

Sobreposição de Métodos (*Override*)

- Tipo de polimorfismo que implica necessariamente em herança, também conhecido como **polimorfismo dinâmico**.
- Ocorre quando uma classe filha **sobrescreve** (ou define) a implementação de um método da classe mãe, porém **sem alterar a assinatura** deste método.
- A decisão de qual método chamar é tomada em **tempo de execução**, com base na instância que chamou o método.
- O polimorfismo dinâmico permite criar métodos flexíveis, que aceitam **argumentos genéricos** e podem receber qualquer tipo pertencente a uma hierarquia de classes.

l.bertholdo@ifsp.edu.br

Sobreposição de Métodos

- Um **método sobrescrito** possui assinatura igual, mas implementação diferente com relação ao método definido na classe mãe.
- Para explicitar que um método está sendo sobrescrito, deve-se usar a anotação **@Override** antes da assinatura do método. Embora não seja obrigatória, seu uso é recomendado.

Exemplo:

Implementação genérica

```
class Funcionario {
    public  double calculaSalario () { salario = salario + vale }
}
```

Implementação específica

```
class Vendedor extends Funcionario {
    @Override
    public double calculaSalario () { salario = salario + vale + comissao }
}
```

Sobreposição de Métodos

- O método da classe mãe, a ser sobrescrito, nunca pode ser **privado**.
- Como uma classe filha não pode aumentar o nível de proteção de um método da classe mãe, seu respectivo método **sobrescrito** na classe filha também não pode ser **privado**.
- Um método sobrescrito pode ser novamente redefinido em outra classe filha (neta com relação à primeira classe mãe).

l.bertholdo@ifsp.edu.br

Sobreposição de Métodos

*Aqui, a decisão de qual método chamar é baseada na **instância** que chamou o método durante a execução.*

Uma variável (ou argumento) do tipo de uma classe mãe pode **instanciar** ou **receber** qualquer tipo de classe filha.

```java
public class Inicio {
    public static void main(String[] args) {
        Mamifero mm = new Mamifero();
        Animal am = new Mamifero();
        Animal aa = new Animal();

        analisarAnimal(mm); // Chama o método "avaliarMetabolismo" da classe filha (Mamifero).
        analisarAnimal(am); // Chama o método "avaliarMetabolismo" da classe filha (Mamifero).
        analisarAnimal(aa); // Chama o método "avaliarMetabolismo" da classe mãe (Animal).
    }

    public static void analisarAnimal(Animal a) {
        a.avaliarMetabolismo();
    }
}
```

```java
public class Animal {
    protected void avaliarMetabolismo(){
        // Código para avaliação de metabolismo de animal.
        System.out.println("Avaliação de metabolismo de animal.");
    }
}
```

```java
public class Mamifero extends Animal {
    // Sobrescreve o método "avaliarMetabolismo" da classe mãe (Animal).
    @Override
    protected void avaliarMetabolismo(){
        // Código para avaliação de metabolismo de mamífero.
        System.out.println("Avaliação de metabolismo de mamífero.");
    }
}
```

```
Avaliação de metabolismo de mamífero.
Avaliação de metabolismo de mamífero.
Avaliação de metabolismo de animal.
```

Sobreposição de Métodos

Note que o **compilador** não tem como saber qual classe este argumento está instanciando (Animal ou Mamifero). Só será possível saber isso **durante a execução**, quando o método for chamado passando um argumento de um tipo específico (Animal ou Mamifero).

```java
public class Inicio {
    public static void main(String[] args) {
        Mamifero mm = new Mamifero();
        Animal am = new Mamifero();
        Animal aa = new Animal();

        analisarAnimal(mm); // Chama o método "avaliarMetabolismo" da classe filha (Mamifero).
        analisarAnimal(am); // Chama o método "avaliarMetabolismo" da classe filha (Mamifero).
        analisarAnimal(aa); // Chama o método "avaliarMetabolismo" da classe mãe (Animal).
    }

    public static void analisarAnimal(Animal a) {
        a.avaliarMetabolismo();
    }
}
```

```java
public class Animal {
    protected void avaliarMetabolismo(){
        // Código para avaliação de metabolismo de animal.
        System.out.println("Avaliação de metabolismo de animal.");
    }
}
```

```java
public class Mamifero extends Animal {
    // Sobrescreve o método "avaliarMetabolismo
    @Override
    protected void avaliarMetabolismo(){
        // Código para avaliação de metabolismo
        System.out.println("Avaliação de metabo
    }
}
```

*O polimorfismo dinâmico permite ter um único método **analisarAnimal**, pois o argumento do método aceita tanto o tipo **Animal** quanto o tipo **Mamifero**, ou qualquer outro tipo que possa ser criado como descendente da classe **Animal**. Sem o polimorfismo seria necessário criar um método **analisarAnimal** para cada tipo de classe de animal (Ave, Réptil, Anfíbio etc).*

Atributos Finais

- Atributos cujos valores não podem ser modificados depois de declarados. Por isso, são utilizados para representar constantes.
- Atributos finais devem ser obrigatoriamente inicializados com um valor.
- Para indicar que um atributo é final, deve-se declará-lo com o operador **final** antes do tipo do atributo.

  Exemplo: private final double pi = 3.14;

Métodos Finais

- Métodos que não podem ser sobrescritos em classes filhas.
- Para indicar que um método é final, deve-se declará-lo com a palavra **final** antes do tipo de retorno do método.
- Um **método sobrescrito** é uma outra implementação de um método de uma classe mãe em uma classe filha.
- Um **método final** é uma outra e última implementação de um método de uma classe mãe em uma classe filha.

l.bertholdo@ifsp.edu.br

Métodos Finais

```java
public class Inicio {
    public static void main(String[] args) {
        Mamifero mm = new Mamifero();
        Animal am = new Mamifero();
        Animal aa = new Animal();

        analisarAnimal(mm); // Chama o método "avaliarMetabolismo" da classe filha (Mamifero).
        analisarAnimal(am); // Chama o método "avaliarMetabolismo" da classe filha (Mamifero).
        analisarAnimal(aa); // Chama o método "avaliarMetabolismo" da classe mãe (Animal).
    }

    public static void analisarAnimal(Animal a) {
        a.avaliarMetabolismo();
    }
}
```

```java
public class Animal {
    protected void avaliarMetabolismo(){
        // Código para avaliação de metabolismo de animal.
        System.out.println("Avaliação de metabolismo de animal.");
    }
}
```

```java
public class Mamifero extends Animal {
    // Sobrescreve o método "avaliarMetabolismo" da classe mãe
    @Override
    protected final void avaliarMetabolismo(){
        // Código para avaliação de metabolismo de mamífero.
        System.out.println("Avaliação de metabolismo de mamífero.");
    }
}
```

A palavra **final** indica que esta é a última implementação deste método, ou seja, ele não pode ser sobrescrito em uma possível classe filha da classe **Mamifero**.

l.bertholdo@ifsp.edu.br

Classes Finais

- Classes que não podem ter classes descendentes, ou seja, não podem herdar seus atributos e métodos para outras classes, nem permitir que seus métodos sejam sobrescritos.

```
public final class Animal {
    protected void avaliarMetabolismo(){
        // Código para avaliação de metabolismo de animal.
        System.out.println("Avaliação de metabolismo de animal.");
    }
}
```

```
public class Mamifero extends Animal {

}
```

l.bertholdo@ifsp.edu.br

Atributos Estáticos

- Atributos que são compartilhados entre todas as instâncias de uma mesma classe. Com isso, o valor de um atributo estático não se altera dependendo da instância que o invoca.
- Para indicar que um atributo é estático, deve-se declará-lo com o operador **static** antes de seu tipo de dado.
- Para invocar um **atributo estático** de fora de sua classe, não se deve usar uma instância. Basta declarar o nome da classe à qual o atributo pertence, seguido de ponto e do nome do atributo. **Exemplo:** Classe.atributo.
- Como não é necessário criar uma instância para invocar um atributo estático, logo a palavra **this** (que representaria essa instância) torna-se desnecessária. Portanto, não se deve usá-la para atributos estáticos.

l.bertholdo@ifsp.edu.br

Atributos Estáticos

Atributo Dinâmico

```
public class Adicao {
    private int soma = 0;

    public void setSoma(int num){
        this.soma = this.soma + num;
    }
    public int getSoma(){
        return this.soma;
    }
}
```

```
Soma: 1
Soma: 3
Soma: 5
```

O valor do atributo **soma** varia conforme a instância que chama o método **getSoma**.

```
public class Inicio {
    public static void main(String[] args) {
        Adicao a = new Adicao();
        a.setSoma(1);
        Adicao b = new Adicao();
        b.setSoma(3);
        Adicao c = new Adicao();
        c.setSoma(5);

        System.out.println("Soma: " + a.getSoma());
        System.out.println("Soma: " + b.getSoma());
        System.out.println("Soma: " + c.getSoma());
    }
}
```

Cada instância referencia um endereço de memória diferente, nos quais se encontram valores distintos.

Atributo Estático

```
public class Adicao {
    private static int soma = 0;

    public void setSoma(int num){
        soma = soma + num;
    }
    public int getSoma(){
        return soma;
    }
}
```

```
Soma: 9
Soma: 9
Soma: 9
```

O valor do atributo **soma** é o mesmo, independentemente da instância que chama o método **getSoma**.

```
public class Inicio {
    public static void main(String[] args) {
        Adicao a = new Adicao();
        a.setSoma(1);
        Adicao b = new Adicao();
        b.setSoma(3);
        Adicao c = new Adicao();
        c.setSoma(5);

        System.out.println("Soma: " + a.getSoma());
        System.out.println("Soma: " + b.getSoma());
        System.out.println("Soma: " + c.getSoma());
    }
}
```

Cada instância referencia um endereço de memória diferente, nos quais se encontram os mesmos valores.

Métodos Estáticos

- Métodos que não possuem vínculos com as instâncias de uma classe.
- São limitados a invocar outros atributos ou métodos **estáticos** da própria classe, ou **dinâmicos** de outras classes.
- Por isso, são usados quando se deseja impedir que o método altere **atributos dinâmicos** ou use **métodos dinâmicos** da própria classe.
- São adequados, por exemplo, para bibliotecas de funções, onde os métodos necessitam apenas dos dados passados como argumentos e, após processá-los, retornam um resultado, não havendo alterações nos atributos dinâmicos da classe.
- Para indicar que um método é estático, deve-se declará-lo com o operador **static** antes do tipo de retorno do método.

l.bertholdo@ifsp.edu.br

Métodos Estáticos

- Para chamar um **método estático** de fora de sua classe, não se deve usar uma instância. Basta declarar o nome da classe à qual o método pertence, seguido de ponto (.) e do nome do método. **Exemplo:** Classe.metodo().
- Como não é necessário criar uma instância para chamar um método estático, logo a palavra **this** (que representaria essa instância) torna-se desnecessária. Portanto, não se deve usá-la para métodos estáticos.
- Uma classe também pode ser estática, desde que ela seja uma **classe interna**.

```
public class Calculo {
    public static class Operacao {
        // Uma classe estática pode ter tanto atributos
        // e métodos estáticos quanto dinâmicos.
    }
}
```

Uma **classe interna** tem acesso a todos os atributos e métodos da **classe externa**, inclusive aos privados.

l.bertholdo@ifsp.edu.br

Métodos Estáticos

Método Dinâmico

```
public class Adicao {
    private int soma = 5;

    public void setSoma(int num) {
        this.soma = this.soma + num;
    }
    public int getSoma() {
        return this.soma;
    }
}
```

```
public class Inicio {
    public static void main(String[] args) {
        Adicao a = new Adicao();
        a.setSoma(3);
        System.out.println("Soma: " +
                           a.getSoma());
    }
}
```

Método Estático

```
public class Adicao {
    private static int soma = 5;

    public static void setSoma(int num) {
        soma = soma + num;
    }
    public static int getSoma() {
        return soma;
    }
}
```

Métodos estáticos podem invocar apenas atributos estáticos e outros métodos estáticos.

```
public class Inicio {
    public static void main(String[] args) {
        Adicao.setSoma(3);
        System.out.println("Soma: " +
                           Adicao.getSoma());
    }
}
```

l.bertholdo@ifsp.edu.br

Classes Abstratas

- Tipo de polimorfismo onde um método de uma classe mãe pode ter diferentes implementações em suas classes filhas, sem que exista uma implementação genérica deste método na classe mãe.

- Neste tipo de polimorfismo, é necessário criar uma **classe mãe abstrata**, cujos métodos abstratos poderão ser implementados de diferentes formas em suas classes filhas.

- Uma classe abstrata pode ter tanto métodos **abstratos** (sem implementação) quanto **concretos** (com implementação).

- Ao herdar de uma classe abstrata, a classe filha necessariamente deve implementar todos os métodos abstratos desta classe.

l.bertholdo@ifsp.edu.br

Classes Abstratas

- A ideia é que um método abstrato seja sempre sobrescrito. Portanto, ele nunca poderá ser **privado**.
- Para indicar que uma classe é abstrata, deve-se usar a palavra-chave **abstract** antes da palavra **class**.
- Para indicar que um método é abstrato, deve-se usar a palavra-chave **abstract** antes do tipo de retorno do método.

```
abstract class ClasseAbstrata{
        protected abstract void metodoAbstrato();
}
```

l.bertholdo@ifsp.edu.br

Classes Abstratas

- O método da classe filha que implementa o método abstrato da classe mãe deve ter em sua assinatura a anotação **@Override**. Contudo, esta anotação não é obrigatória. Exemplo:

```
class ClasseFilha extends ClasseAbstrata {
    @Override
    protected void metodo() {
        // Código do método
    }
}
```

- As assinaturas do método **abstrato** e do respectivo método **implementado na classe filha** devem ser idênticas, ou seja, devem ter o mesmo tipo de retorno, nome, número e tipos de argumentos.
- Além disso, o nível de proteção do método não pode ser aumentado na classe filha, por exemplo, de **protected** para **private**.

Classes Abstratas

- Não é possível criar uma instância do tipo de uma classe abstrata que referencie esta mesma classe abstrata.

  Exemplo: ClasseAbstrata instancia = new ClasseAbstrata();

- Assim, para invocar os **métodos concretos** de uma classe abstrata ou seus **métodos abstratos** (implementados em sua classe filha), é preciso criar uma instância que referencie a classe filha.

  Exemplo: ClasseAbstrata instancia = new ClasseFilha(); ou
  ClasseFilha instancia = new ClasseFilha();

Classes Abstratas

```java
public class Inicio {
    public static void main(String[] args) {
        new Casa(); // Chama o construtor da classe Casa.
        new Apartamento(); // Chama o construtor da classe Apartamento.
    }
}
```

```java
public abstract class Imovel {
    private String endereco;
    private double valor;

    protected void setEndereco(String endereco){
        this.endereco = endereco;
        System.out.println("Endereço cadastrado: " + this.endereco);
    }

    protected void setValor(double valor) {
        this.valor = valor;
        System.out.println("Valor cadastrado: " + this.valor);
    }

    protected abstract void calcularAluguel(double aluguel);
}
```

Classes Abstratas

```java
public class Apartamento extends Imovel {
    private int andar;
    private double aluguel, condominio;

    public Apartamento(){
        super.setEndereco("Rua Itu, 301");
        super.setValor(300000);
        this.setAndar(7);
        this.calcularAluguel(1000);
    }

    public void setAndar(int andar){
        this.andar = andar;
        System.out.println("Andar cadastrado: " + this.andar);
    }

    // Sobrescreve o método CalcularAluguel da classe Imovel.
    @Override
    protected void calcularAluguel(double aluguel){
        this.aluguel = aluguel;
        this.condominio = this.aluguel * 0.25;
        this.aluguel = this.aluguel + this.condominio;
        System.out.println("Aluguel do apto: " + this.aluguel);
    }
}
```

```java
public class Casa extends Imovel{
    private int metroQuadrado;
    private double aluguel;

    public Casa(){
        super.setEndereco("Rua Manaus, 220");
        super.setValor(450000);
        this.setAreaTerreno(250);
        this.calcularAluguel(1500);
    }

    public void setAreaTerreno(int metroQuadrado){
        this.metroQuadrado = metroQuadrado;
        System.out.println("Área cadastrada: " + this.metroQuadrado);
    }

    // Sobrescreve o método CalcularAluguel da classe Imovel.
    @Override
    protected void calcularAluguel(double aluguel){
        this.aluguel = aluguel;
        System.out.println("Aluguel da casa: " + this.aluguel);
    }
}
```

l.bertholdo@ifsp.edu.br

Interfaces

- Interfaces têm algumas similaridades com as classes abstratas, porém há diferenças:
  - Interfaces podem ter apenas atributos **públicos**, **estáticos** e **finais**.
  - Nas interfaces, os métodos também são obrigatoriamente **públicos**.
  - Além de públicos, os métodos não podem ter implementação, ou seja, devem ser **abstratos**, exceto se o método for **estático**.

Interfaces

- A declaração de interfaces é similar à das classes, porém usa-se a palavra-chave **interface** em vez de **class**.

```
interface IFuncionario{
        void calcularVencimento();
}
```

- Para indicar que uma classe implementa uma interface, após o nome da classe, deve-se incluir a palavra **implements** seguida pelo nome da interface. Exemplo:

```
class Atendente implements IFuncionario {
        public void calcularVencimento(){
                // Código do método
        }
}
```

Interfaces

- Não é possível criar uma instância do tipo de uma interface que referencie esta mesma interface.

  Exemplo: Interface instancia = new Interface();

- Assim, para invocar **métodos abstratos** (escritos na classe que implementa a interface), é preciso criar uma instância que referencie a classe que implementa esta interface.

  Exemplo: Interface instancia = new ClasseQueImplementaInterface(); ou ClasseQueImplementaInterface instancia = new ClasseQueImplementaInterface();

Interfaces

- Um classe pode implementar mais de uma interface. Nesse caso, é necessário separá-las por vírgula na declaração da classe. **Exemplo:**
  class Classe implements Interface1, Interface2, Interface3

- Ao implementar uma interface, a classe precisa obrigatoriamente conter todos os métodos abstratos desta interface, ainda que sem implementação.

- O uso de interfaces visa deixar o código mais flexível e possibilitar alterações na implementação sem maiores dificuldades.

Interfaces

```
public interface IFuncionario {
    double calcularVencimento();
}
```

```
public class AuxAdm implements IFuncionario {
    private double vencimento;
    private double salario = 1200;
    private double vale_alim = 200;

    public double calcularVencimento(){
        vencimento = salario + vale_alim;
        return vencimento;
    }
}
```

```
public class Vendedor implements IFuncionario {
    private double vencimento;
    private double salario = 1200;
    private double vale_alim = 200;
    private double comissao = 300;

    public double calcularVencimento(){
        vencimento = salario + vale_alim + comissao;
        return vencimento;
    }
}
```

```
public class Gerente implements IFuncionario {
    private double vencimento;
    private double salario = 3000;
    private double vale_alim = 200;
    private double comissao = 400;
    private double gratificacao = 500;

    public double calcularVencimento(){
        vencimento = salario + vale_alim +
                     comissao + gratificacao;
        return vencimento;
    }
}
```

Interfaces

Para o método **efetuarPagto** não importa de que tipo é o funcionário, ele apenas tem que chamar o método **calcularVencimento**.

```
public class Pagamento {
    public void efetuarPagto(IFuncionario f){
        // Chama o método calcularVencimento da classe instanciada em f.
        System.out.println("O vencimento do funcionário é: " + f.calcularVencimento());
    }
}
```

```
public class Inicio {
    public static void main(String[] args) {
        Pagamento p = new Pagamento();  // Cria uma instância p para a classe Pagamento.

        IFuncionario f = new AuxAdm();  // Cria uma instância f para a classe AuxAdm.
        p.efetuarPagto(f);

        f = new Vendedor();  // f passa a instanciar a classe Vendedor.
        p.efetuarPagto(f);

        f = new Gerente();  // f passa a instanciar a classe Gerente.
        p.efetuarPagto(f);
    }
}
```

Ao criar uma instância do tipo **IFuncionario**, pode-se referenciar qualquer classe que implemente a interface **IFuncionario** (AuxAdm, Vendedor ou Gerente).

Se um dia surgir um novo tipo de funcionário, basta criar uma classe para ele (que implemente a interface **IFuncionario**) e uma instância **IFuncionario** (que referencie esta nova classe). As classes Pagamento, AuxAdm, Vendedor, Gerente e a interface IFuncionario não precisarão sofrer qualquer alteração.

Referências

- Hélio Engholm Jr.; Análise e Design Orientado a Objetos. São Paulo: Novatec Editora, 2013.

- Rafael Santos; Introdução à Programação Orientada a Objetos usando Java – 2ª edição. Rio de Janeiro: Elsevier, 2013.