



# Características das Linguagens de Programação I

**Conteúdo:** Polimorfismo e classe abstrata

Prof. Dsc. Giomar Sequeiros  
[giomar@eng.uerj.br](mailto:giomar@eng.uerj.br)



# **Polimorfismo**

# Polimorfismo

---

- A palavra polimorfismo vem do grego *poli morfos* e significa *muitas formas*. Na orientação a objetos, isso representa uma característica que permite que *classes diferentes sejam tratadas de uma mesma forma*.
- Em outras palavras, podemos ver o *polimorfismo* como a possibilidade de um *mesmo método ser executado de forma diferente* de acordo com a classe do objeto que aciona o método e com os parâmetros passados para o método.

# Polimorfismo

---

- O polimorfismo pode ser obtido pela **utilização** dos conceitos de herança, **sobrecarga** de **métodos** e **sobrescrita** de **método** (também conhecida como redefinição ou reescrita de método).

# Polimorfismo: sobrescrita

- A técnica de sobrescrita permite **reescrever** um método em uma **subclasse** de forma que tenha **comportamento diferente** do método de mesma assinatura existente na sua superclasse.

```
public class Conta {  
  
    public void imprimirTipoConta() {  
        System.out.println("Conta Comum");  
    }  
}
```

```
public class ContaEspecial extends Conta {  
  
    @Override  
    public void imprimirTipoConta() {  
        System.out.println("Conta Especial");  
    }  
}
```

```
public class ContaPoupanca extends Conta {  
  
    @Override  
    public void imprimirTipoConta() {  
        System.out.println("Conta Poupança");  
    }  
}
```



# Polimorfismo: sobrescrita

---

- Nos métodos *ImprimirTipoConta* das classes *ContaEspecial* e *ContaPoupança*, há uma notação **@Override**.
- **A notação @Override é inserida automaticamente pelo IDE** para indicar que esse método foi definido no ancestral e está sendo redefinido na classe atual.
- 
- A não colocação da notação **Override não** gera erro, mas gera um aviso (*Warning*). Isso ocorre porque entende-se que, quando lemos uma classe e seus métodos, é importante existir alguma forma de sabermos se um certo método foi ou não definido numa classe ancestral.
- Assim a notação **@Override é fundamental para aumentar a legibilidade e** manutenibilidade do código.

# Polimorfismo: sobrescrita

- Usando os métodos polimórficos

```
package banco;
import java.util.Scanner;

public class UsaContaPolimorfa {
    public static void main(String[] args) {
        Conta c = null;
        Scanner scan = new Scanner(System.in);
        int opcao;
        System.out.println("Qual tipo de conta deseja criar para José?");
        System.out.println("1 - Conta");
        System.out.println("2 - Conta especial");
        System.out.println("3 - Conta poupança");
        opcao = scan.nextInt();
        switch (opcao) {
            case 1:
                c = new Conta(1, "José");
                break;
            case 2:
                c = new ContaEspecial(1, "José", 100.00);
                break;
            case 3:
                c = new ContaPoupanca(1, "José");
                break;
        }
        c.imprimirTipoConta();
    }
}
```

# Polimorfismo: sobrescrita

- O código mostra método sacar na classe **ContaEspecial** que sobrescreve o método da superclasse e permite a realização do saque caso o valor a ser sacado seja menor ou igual a soma entre o saldo e o limite da conta.

```
@Override
public boolean sacar(double valor){
    if (valor <= this.limite + this.saldo) {
        this.saldo -= valor;
        return true;
    } else {
        return false;
    }
}
```



# Polimorfismo: Sobrecarga

---

- Métodos de mesmo nome podem ser declarados na mesma classe, contanto que tenham diferentes conjuntos de parâmetros (determinado pelo número, tipos e ordem dos parâmetros). Isso é chamado sobrecarga de método
- Para que os métodos de mesmo nome possam ser distinguidos, eles **devem possuir assinaturas diferentes**.
- A **assinatura** (signature) de um método é **composta** pelo **nome** do método e por uma **lista** que indica os tipos de todos os seus **argumentos**. Assim, métodos com mesmo nome são considerados diferentes se recebem um diferente número de argumentos ou tipos diferentes de argumentos e têm, portanto, uma assinatura diferente.

# Polimorfismo: Sobrecarga

- Para ilustrar melhor o conceito de sobrecarga, implementaremos na classe **Conta** um novo método *imprimirTipoConta* que receberá como parâmetro uma String e imprimirá na tela o tipo da conta seguido pela String recebida.

```
public void imprimirTipoConta() {  
    System.out.println("Conta Comum");  
}  
public void imprimirTipoConta(String s) {  
    System.out.println("Conta Comum - String recebida:" + s);  
}
```

# Polimorfismo: Sobrecarga

- A figura exhibe tanto o código-fonte de uma classe que utiliza as classes **Conta** e **ContaEspecial**.

```
package banco;

public class UsaSobrecargaSobrescrita {

    public static void main(String[] args) {
        Conta c1 = new Conta(1, "Ze");
        ContaEspecial c2 = new ContaEspecial(2, "João", 100);
        c1.imprimirTipoConta();
        c1.imprimirTipoConta("Teste Sobrecarga");
        c2.imprimirTipoConta();
        c2.imprimirTipoConta("Teste Sobrecarga em subclasse");
    }
}
```

# Classe abstrata

# Classe abstrata

---

- Uma classe define as características e o comportamento de um conjunto de objetos. Assim, os objetos são criados (instanciados) a partir de classes.
- Mas, nem todas as classes são projetadas para permitir a criação de objetos. Algumas classes são usadas apenas para agrupar características comuns a diversas classes e, então, ser herdada por outras classes. Tais classes são conhecidas como classes abstratas.



# Classe abstrata

---

- As **classes que não são abstratas** são conhecidas como **classes concretas**.
- As **classes concretas** podem ter **instâncias diretas**, ao contrário das **classes abstratas** que só podem ter **instâncias indiretas**, ou seja, apesar de a classe abstrata não poder ser instanciada, ela deve ter subclasses concretas que por sua vez podem ser instanciadas.

# Classe abstrata: Exemplo

- Para ilustrar o conceito de classe abstrata, consideremos um exemplo de contas bancárias. Temos as classes **ContaEspecial** e **ContaPoupança** herdadas da classe **Conta**.
- Agora, suponha que toda conta criada no nosso banco tenha que ser uma conta especial ou uma conta poupança. Nesse caso, nunca teríamos uma instância da classe *Conta*, pois toda conta criada seria uma instância de **ContaEspecial** ou de **ContaPoupança**.
- Nesse contexto surgem algumas perguntas: teria sentido criar a classe *Conta*? Por que criar uma classe que nunca será instanciada?
- A resposta à primeira pergunta é sim! A classe **Conta** *continuará existindo* para organizar as características comuns aos dois tipos de contas. Então, para **garantir que a classe *Conta* exista, mas nunca seja instanciada**, essa classe deve ser **criada como abstrata**.

# Classe abstrata: Exemplo

- Para definir uma classe abstrata em Java, basta utilizar a palavra reservada **abstract**. A palavra `abstract` deve ser inserida entre o qualificador de acesso e o nome da classe.

```
package banco;

public abstract class Conta {
    private int numero;
    private String nome_titular;
    protected double saldo;

    public Conta(int numero, String nome_titular, double saldo) {
        this.numero = numero;
        this.nome_titular = nome_titular;
        this.saldo = saldo;
    }
    public Conta(int numero, String nome_titular) {
        this.numero = numero;
        this.nome_titular = nome_titular;
        saldo = 0;
    }
    public abstract boolean sacar(double valor);
}
```

# Classe abstrata: Exemplo

- Vale ressaltar que a transformação de uma classe em abstrata não traz impacto para nenhum de seus métodos e nem para os códigos das suas subclasses.
- Ao tentar utilizar um construtor de uma classe abstrata para instanciar um objeto, o que acontece? A resposta é **erro de compilação**.

```
1 package banco;
2
3 public class UsaClasseAbstrata {
4
5     public static void main(String[] args) {
6         Conta c1 = new Conta(1, "Ze"); //ERRO!
7         ContaEspecial c2 = new ContaEspecial(2, "João", 100);
8         c1 = new ContaPoupanca(1, "Ze");
9     }
```

# Métodos abstratos

---

- Em algumas situações as **classes abstratas** podem ser utilizadas para **prover a definição de métodos** que devem ser implementados em todas as suas subclasses, sem apresentar uma implementação para esses métodos. Tais métodos são chamados de **métodos abstratos**.
- Para definir um método abstrato em Java, utiliza-se a palavra reservada **abstract** entre o especificador de visibilidade e o tipo de retorno do método. Vale ressaltar que um **método abstrato não tem corpo**, ou seja, apresenta apenas uma **assinatura**.



# Métodos abstratos

---

- No exemplo das contas bancárias. Todo tipo de conta bancária deve ter uma forma de sacar. Mas, de acordo com o tipo da conta, há regras diferentes para o saque.
- Em nosso exemplo, a **ContaEspecial** possui um **limite** de forma que ela permite saques acima do saldo disponível até o limite da conta. Já a **ContaPoupanca** não permite saques acima do saldo disponível.
- Adicionar o método abstrato na classe **Conta**.

```
public abstract boolean sacar(double valor);
```

# Métodos abstratos: Exemplo

- Implementação do método *sacar* na classe **ContaEspecial**

```
public class ContaEspecial extends Conta {  
  
    private double limite;  
  
    @Override  
    public boolean sacar(double valor) {  
        if (valor <= this.limite + this.saldo) {  
            this.saldo -= valor;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

# Métodos abstratos: Exemplo

- Implementação do método *sacar* na classe **ContaPoupanca**

```
public class ContaPoupanca extends Conta {  
  
    @Override  
    public boolean sacar(double valor) {  
        if (this.getSaldo() >= valor) {  
            this.saldo -= valor;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

# Classe abstrata: Exemplo 1

- Criar a classe **Eletrodomestico**

```
public abstract class Eletrodomestico {  
  
    private boolean ligado;  
    private int voltagem;  
  
    public abstract void ligar(); // métodos abstrato ligar  
  
    public abstract void desligar(); // métodos abstrato desligar  
}
```

# Classe abstrata: Exemplo 1 (cont.)

- Adicionando um construtor à classe **Eletrrodomestico**.
- Classes abstratas podem conter métodos construtores, porém não podem ser instanciados diretamente

```
public abstract class Eletrrodomestico {  
  
    private boolean ligado;  
    private int voltagem;  
  
    public abstract void ligar(); // métodos abstrato ligar  
  
    public abstract void desligar(); // métodos abstrato desligar  
  
    // método construtor //  
  
    public Eletrrodomestico(boolean ligado, int voltagem) {  
        this.ligado = ligado;  
        this.voltagem = voltagem;  
    }  
}
```



# Classe abstrata: Exemplo 1 (cont.)

- Classes abstratas podem possuir métodos não abstratos

```
public Eletrodomestico(boolean ligado, int voltagem) {  
    this.ligado = ligado;  
    this.voltagem = voltagem;  
}  
// métodos concretos da classe abstrata  
  
public void setVoltagem(int voltagem) {  
    this.voltagem = voltagem;  
}  
  
public int getVoltagem() {  
    return this.voltagem;  
}  
  
public void setLigado(boolean ligado) {  
    this.ligado = ligado;  
}  
  
public boolean isLigado() {  
    return ligado;  
}  
}
```

# Classe abstrata: Exemplo 1 (cont.)

- Criar a classe **TV**

```
public class TV extends Eletrodomestico {  
    private int tamanho;  
    private int canal;  
    private int volume;  
  
    public TV(int tamanho, int voltagem) {  
        super(false, voltagem); // construtor classe abstrata  
        this.tamanho = tamanho;  
        this.canal = 0;  
        this.volume = 0;  
    }  
}
```

# Classe abstrata: Exemplo 1 (cont.)

- Implementando os métodos abstratos na classe **TV**

```
public TV(int tamanho, int voltagem) {  
    super(false, voltagem); // construtor classe  
    this.tamanho = tamanho;  
    this.canal = 0;  
    this.volume = 0;  
}
```

```
public void desligar() {  
    super.setLigado(false);  
    setCanal(0);  
    setVolume(0);  
}
```

```
public void ligar() {  
    super.setLigado(true);  
    setCanal(3);  
    setVolume(25);  
}
```

# Classe abstrata: Exemplo 1 (cont.)

- Implementando os getters e setters na classe **TV**

```
public void ligar() {  
    super.setLigado(true);  
    setCanal(3);  
    setVolume(25);  
}  
  
public int getTamanho() {  
    return tamanho;  
}  
  
public void setTamanho(int tamanho) {  
    this.tamanho = tamanho;  
}  
  
public int getCanal() {  
    return canal;  
}  
  
public void setCanal(int canal) {  
    this.canal = canal;  
}  
  
public int getVolume() {  
    return volume;  
}  
  
public void setVolume(int volume) {  
    this.volume = volume;  
}
```

# Classe abstrata: Exemplo 1 (cont.)

- Implementando um teste para verificar se a classe **TV** está funcionando

```
public static void main(String[] args) {  
  
    TV minhatv = new TV(29, 127);  
  
    minhatv.ligar();  
  
    System.out.println("Minha TV de " + minhatv.getTamanho() + " polegadas");  
    System.out.print("Neste momento a TV está ");  
    System.out.print(minhatv.isLigado() ? "ligada" : "desligada");  
    System.out.println(minhatv.isLigado() ? " na tensão de " + minhatv.getVoltagem() : " ");  
  
    minhatv.desligar();  
  
    System.out.println("Minha TV " + minhatv.getTamanho() + " polegadas");  
    System.out.print("Neste momento a TV está ");  
    System.out.print(minhatv.isLigado() ? "ligada" : "desligada");  
    System.out.println(minhatv.isLigado() ? " na tensão de " + minhatv.getVoltagem() : " ");  
  
}
```



# Classe final

---

- Uma **classe final não pode ser estendida**.
  - Mecanismo utilizado para impedir a implementação de desdobramentos não planejados em uma biblioteca
- Em Java, utilize o modificador **final** para indicar que a classe é final

```
public final class Integer extends Number {  
    ...  
}
```

- Assim como outras classes da Java API a classe `java.lang.Integer` não pode ser estendida.

# Referências

## BIBLIOGRAFIA BÁSICA:

- DEITEL, Harvery M.. Java : como programar. 10ª ed. São Paulo: Pearson - Prentice Hall, 2017.
- BORATTI, Isaías Camilo. Programação Orientada a Objetos em Java : Conceitos Fundamentais de Programação Orientada a Objetos. 1ª ed. Florianópolis: VisualBooks, 2007.
- SIERRA, Kathy; BATES, Bert. Use a Cabeça! Java. 2ª ed. Rio de Janeiro: Alta Books, 2007.

