Este material foi construído pelo Professor **Valdemar Lorenzon Junior** (Lorenzon), trazendo adaptações de material de própria autoria ao longo de anos de dedicação e trabalhos com pesquisa, desenvolvimento e docência com OO, recuperação de cursos certificados SUN e Oracle.

É um material teórico/prático enriquecido com atualizações de conteúdo e certificações de updates, podendo trazer alguma incompatibilidade em algum momento que pode ser corrigida quando observada.

Sumário

Polimorfismo

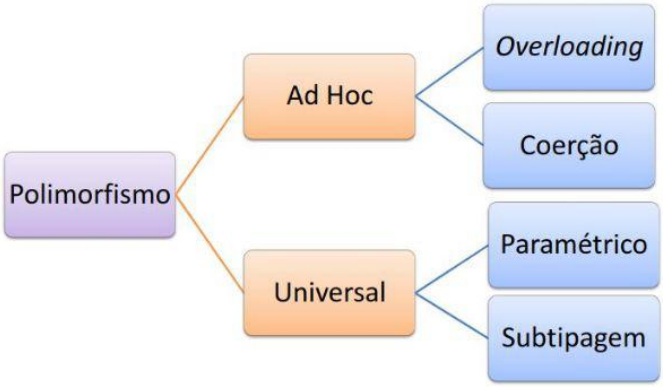
O polimorfismo é um dos pilares fundamentais da orientação a objetos, destacando-se como **a capacidade de objetos diferentes responderem de maneiras distintas a uma mesma mensagem**, **de acordo com sua especialização** individual.

Isso significa que, mesmo compartilhando uma interface comum, cada objeto pode oferecer implementações específicas de comportamento, adaptando-se de forma flexível às necessidades do programa em tempo de execução.

Em termos simples, o polimorfismo permite que objetos de diferentes classes sejam tratados de forma uniforme, desde que essas classes estejam relacionadas por herança ou implementação de interfaces.

O polimorfismo facilita a criação de código mais flexível e extensível, permitindo que você escreva programas mais eficientes e fáceis de manter. Ele também promove o reuso de código, já que você pode usar métodos comuns para manipular objetos de diferentes classes.

Podemos observar atualmente em JAVA vários tipos de polimorfismo, muitas formas, aplicados na linguagem:



O polimorfismo ad hoc permite que diferentes tipos de objetos sejam tratados de maneira polimórfica com base em suas capacidades individuais, sem a necessidade de uma hierarquia de classes ou interfaces comuns.

- **Polimorfismo (Sobrecarga - Overloading):** Este tipo de polimorfismo permite que você defina múltiplos métodos com o mesmo nome em uma classe, mas com diferentes assinaturas de parâmetros. O método a ser chamado é decidido em tempo de compilação com base nos tipos e na quantidade de argumentos passados para o método;

- **Polimorfismo de Coerção (Casting - Subtype - Coercion):** Este tipo de polimorfismo envolve a conversão de um tipo de dado em outro tipo compatível. Isso pode ser feito implicitamente ou explicitamente. A coerção implícita acontece

quando o compilador JAVA converte automaticamente entre tipos primitivos compatíveis. A coerção explícita (casting) é quando você converte explicitamente um tipo de dado em outro, como converter uma referência de uma classe pai para uma classe filha;

O polimorfismo universal, conceitualmente, refere-se à capacidade de um único conceito ou operação poder ser aplicado a diferentes tipos de dados, de forma transparente e sem a necessidade de conhecimento específico sobre o tipo de dado em questão. Simplifica o código, promove a reutilização e aumenta a flexibilidade, tornando-o uma característica valiosa em linguagens de programação orientadas a objetos e funcionais;

- **Polimorfismo de Subtipagem (Subtyping - Inheritance):** Este tipo de polimorfismo ocorre quando uma classe é uma subclasse de outra classe e substitui (override) um ou mais métodos da classe pai. Isso também permite que você trate objetos de subclasses de forma polimórfica, ou seja, usando referências de classe pai para manipular objetos de classe filha.

- **Polimorfismo Universal Paramétrico (Generics**): Este tipo de polimorfismo permite que você escreva código que pode operar em tipos genéricos. Ele oferece flexibilidade ao criar classes e métodos que podem trabalhar com diferentes tipos de dados de forma genérica, sem a necessidade de reescrever o código para cada tipo específico.

Tipo de Polimorfismo	Nome Técnico	Momento da Decisão	Como Funciona	Exemplo Típico	Quando Usar
Sobrecarga	*Overloading*	**Compilação**	Métodos com mesmo nome, mas **assinaturas diferentes** (parâmetros mudam).	soma(int, int) e soma(double, double)	Quando quer **mesma ideia de operação**, mas para **tipos ou quantidades diferentes de dados**.
Sobrescrita	*Overriding*	**Execução**	Classe filha redefine método da classe pai. O comportamento real depende do objeto instanciado.	Cachorro.fazerSom() sobrescreve Animal.fazerSom()	Quando precisa de **especialização de comportamento** em subclasses.
Paramétrico	*Polymorphism with Generics*	**Compilação**	Usa **Generics (<>)** para escrever código que funciona para múltiplos tipos.	Caixa<T> pode armazenar Integer ou String.	Quando quer **reutilizar código** sem duplicar para cada tipo.
De Subtipos	*Subtype Polymorphism*	**Execução**	Um objeto concreto pode ser acessado via **referência genérica** (interface ou superclasse).	Forma f = new Circulo()	Quando deseja **programar para interfaces/abstrações**, aumentando **flexibilidade e baixo acoplamento**.

Polimorfismo de Sobrecarga (Overloading)

Este tipo de polimorfismo ocorre quando uma classe tem métodos com o mesmo nome, mas diferentes parâmetros. O método a ser invocado é determinado pela **assinatura** do método. Por exemplo:

```
class CalculadoraDoisOperadores {

    int somar (int x, int y) {
        return x + y;
    }

    double somar (double x, double y) { //overloading - assinatura
        return x + y;
    }
}

public class App {
    public static void main(String[] args) {
        CalculadoraDoisOperadores calculadora =
                                    new CalculadoraDoisOperadores();

        int    resultado1  = calculadora.somar(3, 4); // Saída: 7
        double  resultado2  = calculadora.somar(3.5, 4.5); // Saída: 8.0

        System.out.println (resultado1);
        System.out.println (resultado2);

    }
}
```

Apenas a mudança de retorno não é válida para sobrecarga.

A sobrecarga pode mudar em retorno sempre em conjunto com a assinatura. Apenas retorno não define a sobrecarga.

Já temos experiência de sobrecarga com múltiplos construtores criados numa classe.

Exercício:

A exemplo da classe calculadora utilizada, complemente a mesma de forma que suporte a soma de 3 operandos e avance de forma que a mesma também faça a operação de multiplicação de 2 e 3 argumentos. Crie um pacote e uma aplicação, dentro dos moldes esperados de aplicação já estudados.

Polimorfismo de Coerção (Casting)

O **polimorfismo de coerção**, também chamado de **coerção de tipo**, acontece quando um objeto é convertido de um tipo para outro **dentro da hierarquia de herança**.

A operação ocorre em em tempo de compilação/execução para **ver um objeto através de outro tipo da hierarquia**.

Em **Java**, isso pode ocorrer de duas formas nas referências:

- **Implícita (upcasting):** conversão automática de um tipo mais específico (subclasse) para um tipo mais genérico (superclasse ou interface). Não exige código extra do programador. Usado quando a programação é para a implementação (ou abstração);
- **Explícita (downcasting):** conversão forçada de um tipo genérico para um tipo mais específico. Exige o uso de **casting** ((Classe) objeto) e deve ser feita com cuidado, pois pode causar erro em tempo de execução se o objeto não for realmente da classe indicada. Usado quando a programação é para o concreto, para a especialização.

Quando o **Java permite automaticamente** que um objeto seja visto como sua **superclasse** ou como uma **interface** que ele implementa. O programador **não precisa escrever nada**, o compilador já aceita.

Considere o seguinte exemplo hierárquico, com o propósito de exibir a coerção:

```
interface Sonoro {
    public void fazerSom();
}



class Animal implements Sonoro {
    @Override
    public void fazerSom() {
        System.out.println("Som genérico de animal!");
    }
}
```

```
class Cachorro extends Animal {
    public void latir() {
        System.out.println("Au au!");
    }
}
class Gato extends Animal {
    public void miar() {
        System.out.println("Miau!");
    }
}
```

```
public class Coercao {
    public static void main(String[] args) {

        //Programando pra implementação
        Sonoro animal1 = new Animal(); //coerção implícita*
        //Programando pra implementação
        Animal animal2 = new Animal();
        Animal animal3 = new Cachorro(); //coerção implícita*
        Gato animal4 = new Gato();

        //Os objetos irão responder de acordo com sua especialização, mas
        //o compilador entende pela referência

        animal1.fazerSom();
        animal2.fazerSom();
        animal3.fazerSom();
```

```
//Porque animal 3 não late, na verdade ele late
        //animal3.latir();
        //assim
        ((Cachorro)animal3).latir();
        if (animal3 instanceof Cachorro){
            ((Cachorro)animal3).latir(); // coerção explícita **
        }

        animal4.fazerSom();
        animal4.miar();

        //E se colocarmos o animal4 em algum lugar?
        animal1 = animal4;
        /*como ele poderia fazer isso
        animal1.fazerSom();
        animal1.miar();*/

        if (animal1 instanceof Gato){
            animal1.fazerSom();
            ((Gato)animal1).miar();
        }

    }
}
```


* Coerção implícita

Cachorro **é um** Animal, então pode ser referenciado por uma variável do tipo Animal. Animal **implementa** Sonoro, então pode ser guardado em uma variável do tipo Sonoro. **O compilador aceita automaticamente**, pois não há perda de informação: todo Cachorro é também um Animal e todo Animal é Sonoro.

** Coerção explícita

É quando você precisa **forçar a conversão** para um tipo mais específico. O compilador não faz isso sozinho, porque pode dar errado em tempo de execução.

animal3 foi declarado como Animal, mas na memória ele é um Cachorro. Para acessar o método específico latir(), que **não existe em Animal**, você precisa **dizer ao compilador**: "Confie em mim, esse objeto é mesmo um Cachorro!"

Por isso fazemos o cast ((Cachorro) animal3). Se o objeto **não fosse de fato um Cachorro**, haveria um erro de execução: ClassCastException.

Pela referência Sonoro, só conseguimos chamar o que a **interface define** fazerSom()). Para acessar miar(), que só existe em Gato, precisamos de coerção explícita.

A classe de uma referência pode ser acessada a partir do operador **instanceof**.

Buscando um exemplo de uso com base nos exemplos citados neste material, podemos exibir um exemplo de uso como segue:

```
if (animal3 instanceof Cachorro){
    ((Cachorro)animal3).latir();
}
```

Exercício:

Seguindo no exemplo dado, crie um pardal (“piu-piu”) e uma galinha (”cócóricó”) como subclasses extras e aplique da mesma forma para cachorro e gato na aplicação objetos desta nova classe.

Polimorfismo de Subtipo (Herança)

O polimorfismo de subtipo é uma relação **conceitual** entre tipos, estabelecida pela herança ou pela implementação de interface. Baseando-se no exemplo anterior:

- Cachorro **é subtipo** de Animal porque **extends Animal**.
- Animal **é subtipo** de Sonoro porque **implements Sonoro**.

É definido no **código da classe** (no extends ou implements). Permite assim **programar para a abstração**: você declara variáveis, parâmetros ou retornos como Animal ou Sonoro e pode receber qualquer subtipo.

Este tipo de polimorfismo ocorre quando uma classe herda de outra classe e substitui um ou mais métodos dessa classe pai. Isso permite que você substitua comportamentos padrão por comportamentos específicos em subclasses.

As referências são upcasting e a construção é downcasting. O construtor então é uma especialização do objeto.

O polimorfismo garantirá que a resposta a mensagem do método estará de acordo com a especialização do objeto, mesmo que sejam todos do tipo super.

Esse recurso é útil para **tratar objetos de subclasses diferentes de maneira uniforme**, permitindo programar para a abstração (interfaces ou classes mais genéricas) e, quando necessário, acessar comportamentos específicos de uma subclasse.

Ligação tardia

A ligação tardia (late binding) é a chave para o funcionamento do polimorfismo universal em Java. O compilador não gera código em tempo de compilação. Cada vez que se invoca um método de um objeto, o compilador gera código para verificar qual método deve ser chamado.

Exemplo:

```
class FormaGeometrica {

    public void Introduz(){

                System.out.print("\nSou uma forma geométrica");

    }

}
```

```
class Retangulo extends FormaGeometrica {

    public void Introduz() {

                System.out.print("\nSou um retângulo");

    }

}
```

```
class Circulo extends FormaGeometrica {

    public void Introduz() {

                System.out.print("\nSou um círculo");

    }

}
```

```
public class TestePolimorfismo {

    public static void main(String [] s)

    {

        FormaGeometrica refFG1, refFG2, refFG3;

        refFG1 = new FormaGeometrica();

        refFG2 = new Retangulo();

        refFG3 = new Circulo();

        refFG1.Introduz();

        refFG2.Introduz();

        refFG3.Introduz();

    }

}
```

Saída do programa:

```
Sou uma forma geométrica

Sou um retângulo

Sou um círculo
```

Polimorfismo em programação orientada a objetos então em parte é fato de objetos diferentes responderem a uma mesma mensagem (i.e., chamada de método) de maneiras diferentes de acordo com a sua especialização!

Lembre-se: Para chamar o método da superclasse usa-se **super.**método(). Isso dentro da classe.

Um exemplo para coerção e subtipo

Um exemplo mais amplo para estes fenômenos pode ser entendido com o programa a seguir que simula um carrinho de compras de um site qualquer (analogia não web).

Neste exemplo, vamos ver uma interface, uma classe abstrata, uma tratativa de exceção, o uso de coleção do tipo ArrayList, ao invés de um arranjo.

Apresentamos um sistema simples de compras em que diferentes aparelhos, como televisores, podem ser adicionados a um carrinho e ter seus preços totalizados.

```
//Autor: Valdemar Lorenzon Junior
import java.util.ArrayList;
import java.util.List;

// Classe de exceção para carrinho vazio
/*public*/ class ExceptionCarrinhoVazio extends Exception {
    public ExceptionCarrinhoVazio() {
        super("Carrinho Vazio");
        System.out.println("Carrinho Vazio");
    }
}

// Interface que obriga todo item vendável a ter um preço
/*public*/ interface ItemVendavel {
    public double getPreco();
}

// Classe pai abstrata para garantir que todo "Aparelho" é um
ItemVendavel
/*public*/ abstract class Aparelho implements ItemVendavel {
    // se não implementar getPreco, será abstrato
}
```

```
// Classes filhas com implementação de preço
/*public*/ class Tv extends Aparelho {
    private double preco = 1050.00;

    @Override
    public double getPreco() {
        return preco;
    }

    // Método específico da TV (exemplo para coerção explícita depois)
    public void ligarSmart() {
        System.out.println("Abrindo Netflix...");
    }
}

/*public*/ class TvCabo extends Aparelho {
    private double preco = 501.00;

    @Override
    public double getPreco() {
        return preco;
    }

    // Método específico da TvCabo
    public void ligarCabo() {
        System.out.println("Sintonizando canais a cabo...");
    }
}
```

```
// Classe Compras que gerencia o carrinho
/*public*/ class Compras {
    private List<ItemVendavel> carrinho; // programando para a interface

    public Compras() {
        carrinho = new ArrayList<>();
    }

    public void addCompra(ItemVendavel item) {
        carrinho.add(item); // coerção implícita (upcasting)
    }

    public double total() throws ExceptionCarrinhoVazio {
        if (carrinho.isEmpty()) {
            throw new ExceptionCarrinhoVazio();
        }

        double t = 0;
        for (ItemVendavel item : carrinho) {
            t += item.getPreco();
        }
        return t;
    }
}
```

```
// Classe principal
/*public*/ public class CoercaoeSubtipo {
    public static void main(String[] args) {
        try {
            Compras minhasCompras = new Compras();
            // Coerção implícita: Tv e TvCabo são subtipos de
ItemVendavel
            minhasCompras.addCompra(new Tv());
            minhasCompras.addCompra(new TvCabo());
            minhasCompras.addCompra(new TvCabo());
            // Descobrindo o total
            try {
                System.out.println("O total: " + minhasCompras.total());
            } catch (ExceptionCarrinhoVazio e) {
                System.out.println("Não há compras no carrinho!");
            }

            // Exemplo de coerção explícita (downcasting)
            ItemVendavel item = new Tv();
            // Não consigo chamar ligarSmart() direto, só getPreco()
            System.out.println("Preço do item: " + item.getPreco());

            // Downcasting para acessar comportamento específico
            if (item instanceof Tv) {
                ((Tv) item).ligarSmart();
            }
        } catch (Exception e) {
            System.out.println("Erro: " + e.getMessage());
        } finally {
            System.out.println("Fim de execução.");
        }
    }}
```

Para garantir que todo item do carrinho possua um preço, foi definida a interface ItemVendavel, que obriga a implementação do método getPreco().

Essa escolha permite programar para a abstração, ou seja, utilizar uma interface genérica como referência em vez de trabalhar diretamente com classes concretas. Assim, o carrinho de compras manipula uma lista de ItemVendavel, sem precisar conhecer previamente se o objeto é uma Tv ou uma TvCabo.

Ao usar a interface como tipo de referência, o código ilustra o polimorfismo de subtipo. Tanto Tv quanto TvCabo são subtipos de ItemVendavel e podem ser tratados uniformemente dentro da coleção. Essa característica permite que objetos de subclasses diferentes sejam manipulados de maneira genérica, facilitando a extensão do sistema caso novos aparelhos sejam adicionados futuramente, sem necessidade de alterar a lógica central de cálculo do total.

O exemplo também mostra o papel da coerção de tipos. Quando uma Tv é atribuída a uma variável do tipo ItemVendavel, ocorre uma coerção implícita (upcasting): o compilador aceita automaticamente porque todo Tv é um ItemVendavel. No entanto, se quisermos acessar um comportamento específico de Tv, como o método ligarSmart(), precisamos realizar uma coerção explícita (downcasting), convertendo a referência genérica de volta para o subtipo concreto. Essa operação exige cuidado e costuma ser acompanhada de verificações com instanceof para evitar erros em tempo de execução.

Dessa forma, o código traz de forma integrada os conceitos de programação para interfaces, polimorfismo de subtipo e coerção de tipos, mostrando como a linguagem Java permite tratar coleções de objetos heterogêneos de forma uniforme e, ao mesmo tempo, acessar comportamentos específicos quando necessário.

1) **A visão do compilador**: O compilador vê TV como um tipo de ItemDeVenda. Por outro lado, todos os métodos que são chamados pelo objeto item devem estar definidos em ItemDeVenda, pois item é um ItemDeVenda (como referência). Essencialmente, você está escrevendo um código genérico para uma comum funcionalidade para qualquer classe derivada de ItemDeVenda;

2) **A visão da JVM**: Ao executar, a JVM criará dinamicamente o objeto item. Quando um método deste for chamado, por exemplo: `getPreco()`, o método modificado polimorficamente é chamado.

Ocultação e Overriding

Overriding

A assinatura de um método consiste de seu nome e dos tipos de seus parâmetros. Quando um método de uma subclasse tem a mesma assinatura e o mesmo tipo de retorno de um método de uma de suas superclasses, diz-se que o método da subclasse predomina sobre o método correspondente na superclasse e este fato em si é denominado predominância ou sobreposição (overriding) de método.

Em Java, não é permitido que um método tenha a mesma assinatura e tipo de retorno diferente de um método de uma de suas superclasses. No entanto, é permitido que um método tenha o mesmo nome e assinatura diferente de um método de uma de suas superclasses. Neste último caso, ocorre sobrecarga de método, conforme já foi discutido antes.

Ocultação

Quando um campo é declarado numa subclasse utilizando o mesmo nome de um campo existente na superclasse, diz-se que o campo na subclasse oculta o campo da superclasse.

Um campo de classe que é oculto numa subclasse pode ser acessado (se lhe for permito acesso, obviamente) apenas qualificando-o com a palavra reservada **super**. Um método de uma superclasse que sofre overriding numa subclasse pode também ser acessado na subclasse usando super.

Lembre-se do uso de **this** para os conflitos de escopo e **super** agora neste contexto.

Polimorfismo com Generics

Polimorfismo com generics em JAVA refere-se à capacidade de criar classes e métodos que podem trabalhar com diferentes tipos de dados de forma genérica. Isso permite escrever código mais flexível e reutilizável, pois não é necessário reescrever o mesmo código para cada tipo de dado que você deseja manipular.

Neste exemplo a seguir, a classe Caixa é uma classe genérica que pode conter qualquer tipo de objeto. O tipo de objeto é especificado entre os símbolos de **diamante <>**. Quando você cria uma instância da classe Caixa, você especifica o tipo de objeto que deseja armazenar nela.

T representa o tipo, genérico dentro da classe.

Ao usar generics, você pode escrever código que é independente do tipo de dados que está sendo manipulado. Isso promove o reuso de código e a flexibilidade, pois você pode usar a mesma classe ou método com diferentes tipos de dados sem precisar reescrevê-los.

O polimorfismo com generics em JAVA é uma poderosa ferramenta para tornar seu código mais genérico e extensível.

Exemplo para ilustrar o polimorfismo com generics em JAVA:

```
class Caixa<T> {
    private T conteudo;

    public void colocar(T objeto) {
        this.conteudo = objeto;
    }

    public T retirar() {
        return conteudo;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Caixa<Integer> caixaInteiro = new Caixa<Integer>();
        caixaInteiro.colocar(10);
        System.out.println
         ("Conteúdo da caixa de inteiros: " +
                caixaInteiro.retirar());

        Caixa<String> caixaString = new Caixa<>();
        caixaString.colocar("Olá, mundo!");
        System.out.println
              ("Conteúdo da caixa de strings: " +
                                    caixaString.retirar());
    }
}
```

Métodos genéricos

Em Java, é possível criar métodos genéricos que utilizam o polimorfismo para trabalhar com diferentes tipos de dados de forma flexível e genérica. Esses métodos genéricos podem ser definidos em classes genéricas ou em classes não genéricas.

Utiliza-se a convenção **<T>** e **T** para tornar o método genérico, independente da classe ser ou não genérica. A seguir está um exemplo de como criar um método genérico em uma classe não genérica

```
public class Utils {

    public static <T> void imprimirElemento(T elemento) {
        System.out.println(elemento);
    }
```

```
        public static void main(String[] args) {
            Integer numero = 10;
            String texto = "Olá, mundo!";
            //Chama o método genérico para imprimir um númerointeiro
            imprimirElemento(numero);
            // Chama o método genérico para imprimir uma string
            imprimirElemento(texto);
    }
    }
```

Neste exemplo, o método **imprimirElemento** é genérico e pode aceitar qualquer tipo de argumento. O tipo do argumento é definido entre os sinais de diamante **<T>.** Dentro do método, **T** é tratado como o tipo genérico, permitindo que você trabalhe com qualquer tipo de dado.

Você pode usar esse método com qualquer tipo de dado, e o compilador Java irá inferir automaticamente o tipo com base no argumento passado para o método.

Se você deseja criar um método genérico em uma classe genérica, a sintaxe é como repassado no exemplo da classe genérica.

Expansões com <T>

<T>: genérico básico.

<T extends Classe> : restringe para subtipos.

<T extends Classe & Interface> : múltiplas restrições.

<?>, <? extends>, <? super> : curingas para mais flexibilidade: qualquer, qualquer que seja subtipo de, qualquer supertipo de;

<> (diamond operator) : simplificação de sintaxe, exemplo: `List<String> lista = new ArrayList<>();`

<T> para armazenamento dinâmico de qualquer tipo de objetos.

```
import java.util.ArrayList;

// Classe genérica para armazenar objetos de qualquer tipo
class Armazenamento<T> {
    private ArrayList<T> lista;
    // Construtor inicializa o ArrayList
    public Armazenamento() {
        lista = new ArrayList<>();
    }

    // Método para adicionar um objeto ao ArrayList
    public void adicionar(T objeto) {
        lista.add(objeto);
    }
    // Método para remover um objeto do ArrayList
    public void remover(T objeto) {
        lista.remove(objeto);
    }
    // Método para obter o tamanho do ArrayList
    public int tamanho() {
        return lista.size();
    }
    // Método para imprimir todos os elementos do ArrayList
    public void imprimir() {
        for (T elemento : lista) { // For each
            System.out.println(elemento); //aciona o toString()
        }
    }
}
```

```
// Exemplo de uso da classe genérica
public class App {
    public static void main(String[] args) {
     // Criando um objeto da classe genérica para armazenar Strings
     Armazenamento<String> armazenamentoStrings = new Armazenamento<>();

     // Adicionando elementos
     armazenamentoStrings.adicionar("Elemento 1");
     armazenamentoStrings.adicionar("Elemento 2");
     armazenamentoStrings.adicionar("Elemento 3");

     // Imprimindo o tamanho do vetor
     System.out.println
          ("Tamanho do vetor: " + armazenamentoStrings.tamanho());

     // Imprimindo todos os elementos do vetor
     System.out.println("Elementos:");
     armazenamentoStrings.imprimir();

     // Removendo um elemento
     armazenamentoStrings.remover("Elemento 2");
     // Imprimindo o tamanho atualizado do vetor
     System.out.println
               ("Tamanho do vetor após remoção: " +
                         armazenamentoStrings.tamanho());
     // Imprimindo todos os elementos do vetor após remoção
     System.out.println("Elementos após remoção:");
     armazenamentoStrings.imprimir();
  }
}
```

Exercício:

Sabendo do uso de ArrayList e Generics, tendo no conteúdo como manipular o mesmo, também utilizando o exemplo de polimorfismo com a superclasse animal e derivações: Cachorro, Gato, Galinha e Pardal. Faça um código, separando cada classe em seu arquivo distinto, declarando um pacote MundoAnimal e por herança vá construindo as classes e derivações.

A aplicação deve criar vários objetos de cada classe e armazenar estes objetos em um objeto de ArrayList.

Por fim, varra o objeto e faça cada item adicionado no objeto da coleção emitir seu som particular.

Observação: varrer uma coleção é passar item a item, desde seu primeiro item até o último. 0 a n -1, onde n é o tamanho em itens do objeto da coleção.