Aula 10

Herança

Programação III

Prof. Augusto César Oliveira
augusto.oliveira@unicap.br

UNIVERSIDADE CATÓLICA DE PERNAMBUCO

Na aula passada…

- Compreender o conceito de `ArrayList`;
- Comparar `ArrayList` com arrays tradicionais e entender as suas vantagens;
- Declarar e inicializar `ArrayLists`;
- Inserir e remover elementos em um `ArrayList`;
- Acessar elementos em um `ArrayList`;
- Iterar sobre um `ArrayList`.

O objetivo da aula de hoje…

- Compreender o conceito de herança e sua aplicação na programação orientada a objetos;
- Compreender o significado e o uso da palavra-chave "final" em relação a classes, métodos e atributos;
- Entender os modificadores de acesso (public, private, protected) e seu papel na encapsulação;

O objetivo da aula de hoje…

- Compreender o conceito de classes abstratas e seu propósito na modelagem de classes;
- Compreender o conceito de polimorfismo e sua importância na programação orientada a objetos.

1. Herança

Herança

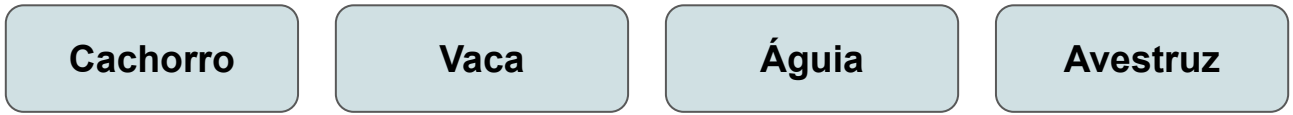
O que é "herança"?

- Princípio que **permite que uma classe compartilhe atributos e métodos com outra classe**, especializando seu estado e comportamento.

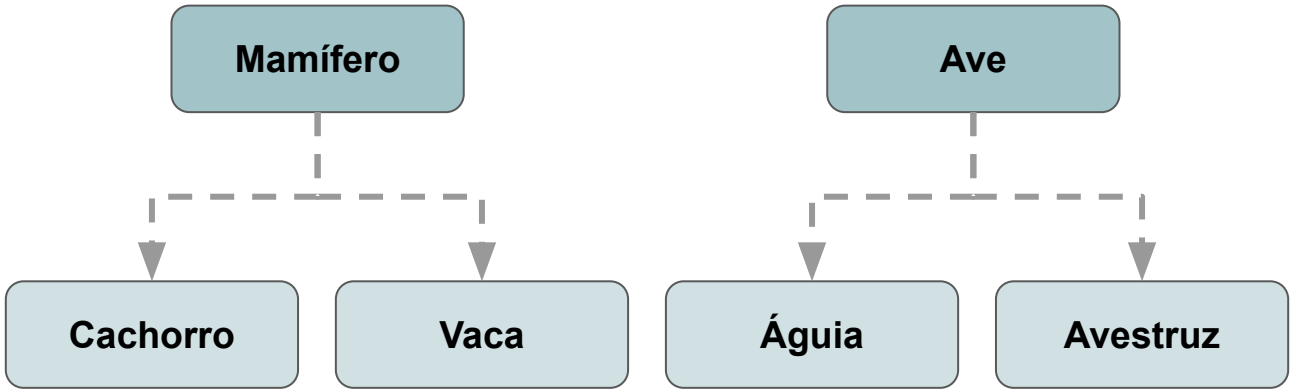
- Vantagens:

    1. Permite o **reuso de código**, reduzindo o retrabalho;
    2. **Evita erros** originados por repetição de código.

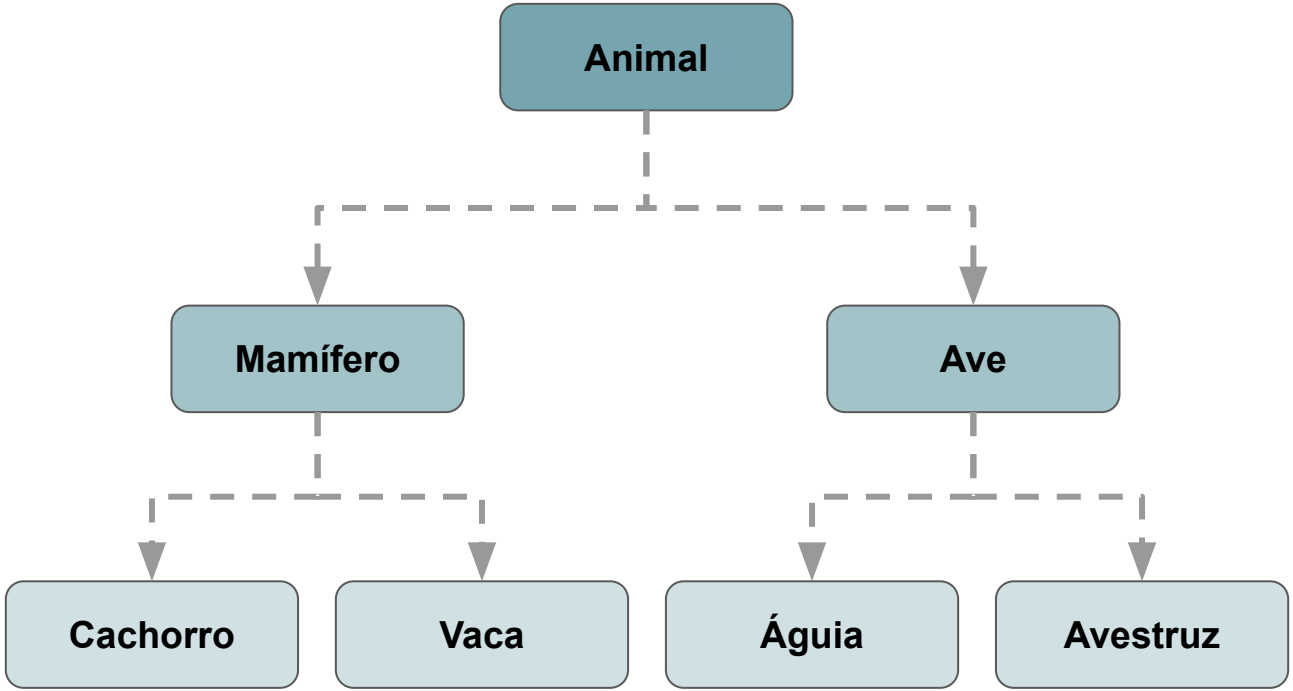
Exemplo: herança



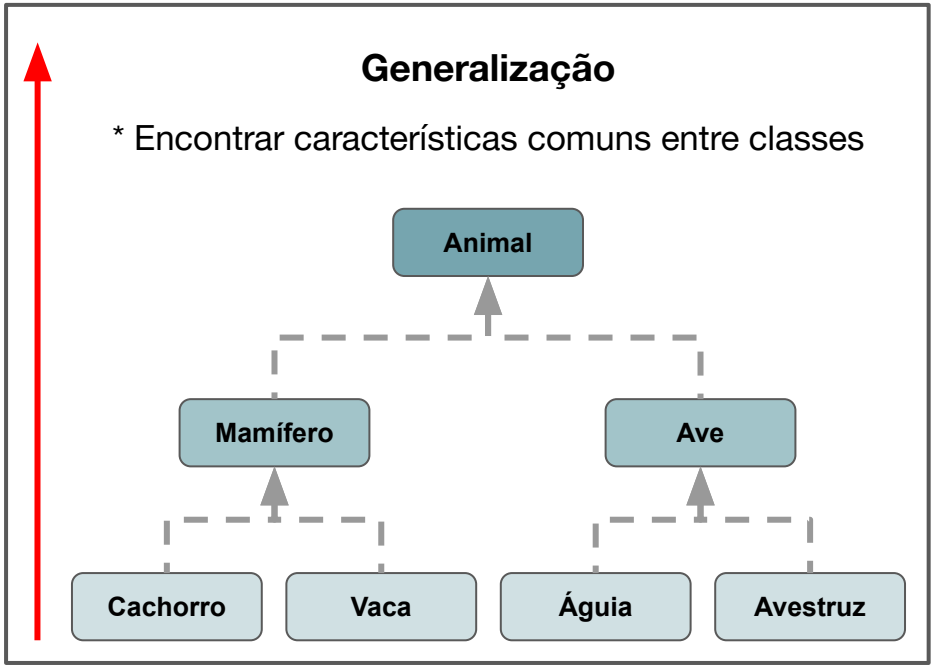
Exemplo: herança

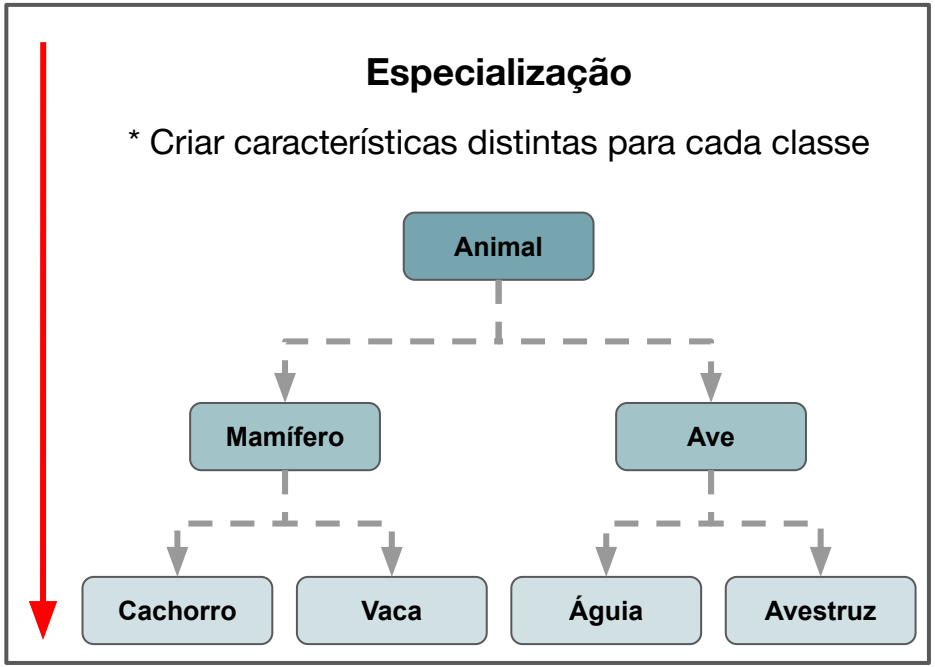


Exemplo: herança

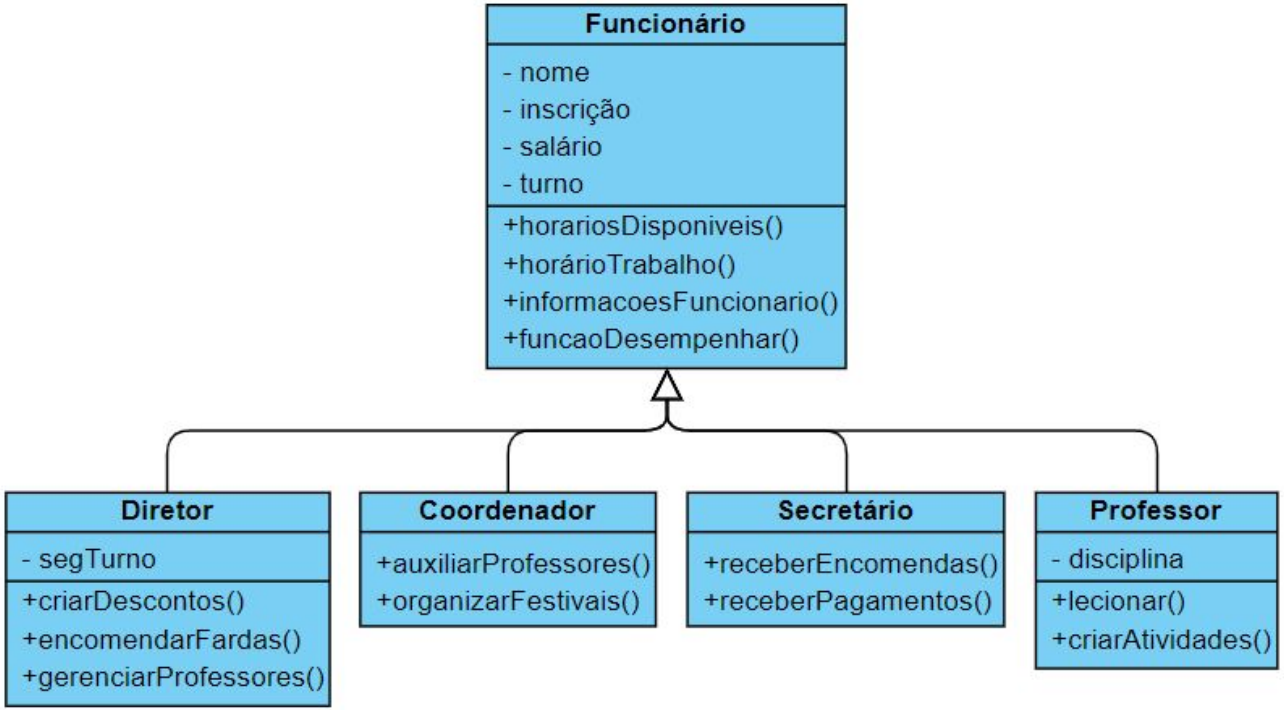


Generalização e especialização



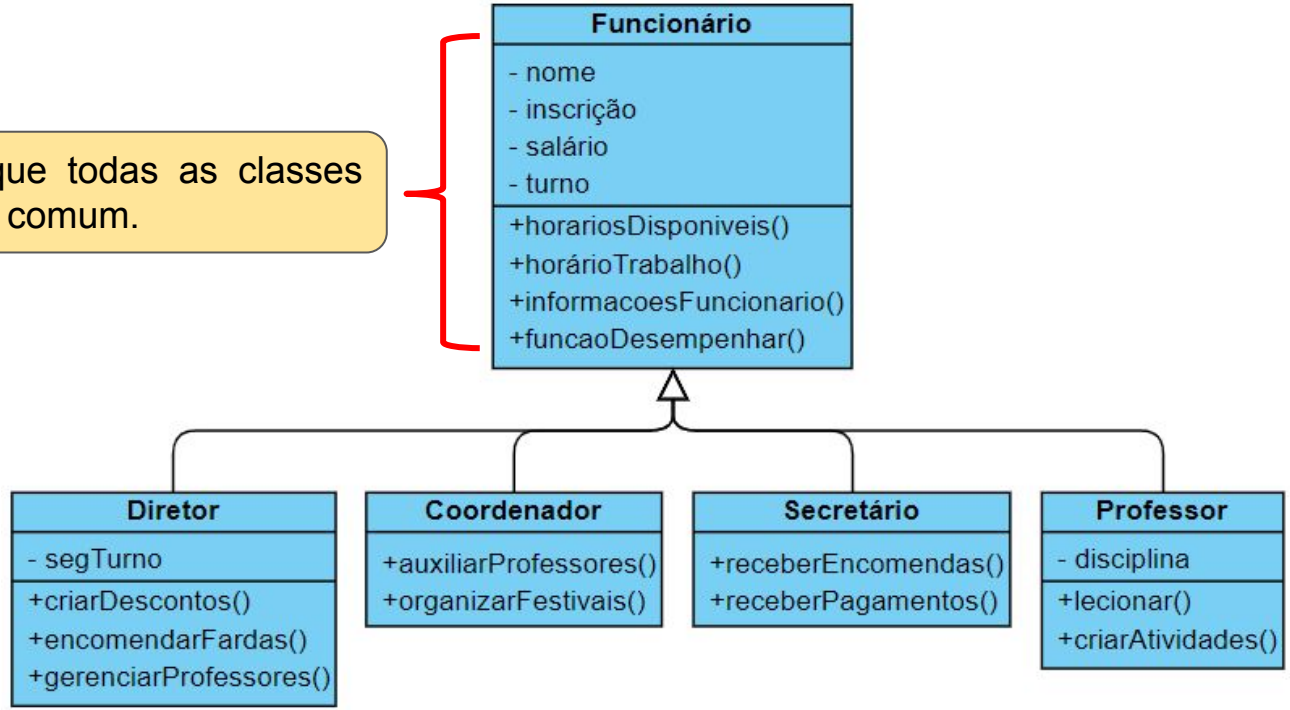


Exemplo: sistema de gerenciamento de funcionários da educação

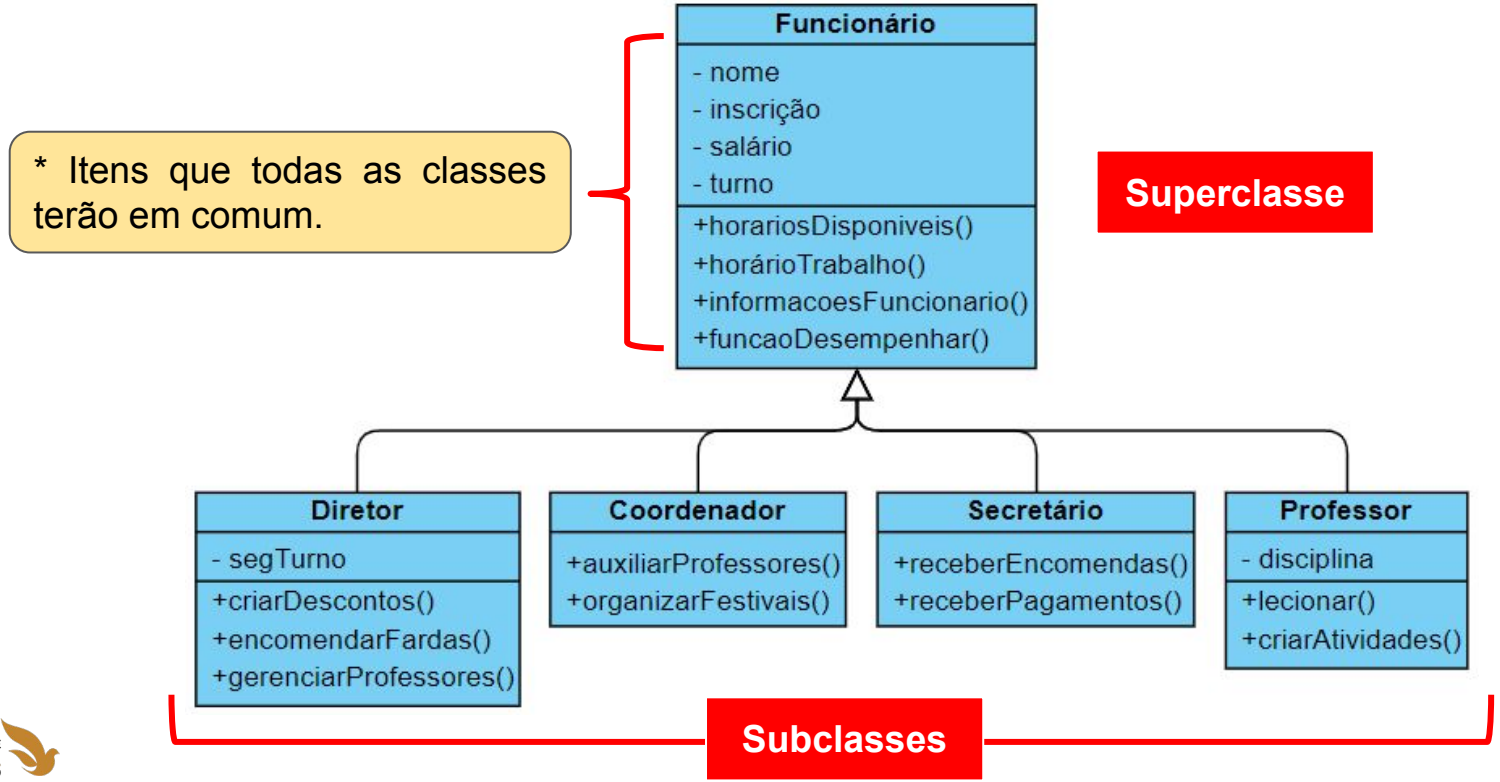


Exemplo: sistema de gerenciamento de funcionários da educação

* Itens que todas as classes terão em comum.



Exemplo: sistema de gerenciamento de funcionários da educação



No contexto de uma empresa…

- Como podemos modelar as classes "**Programador**" e "**Designer**"?

```java
public class Programador {

    private String nome;
    private String cpf;
    private double salario;
    private String linguagem;

    // Métodos GET e SET...
}
```

```java
public class Designer {

    private String nome;
    private String cpf;
    private double salario;
    private String softwareEdicao;

    // Métodos GET e SET...
}
```

O que as classes têm em comum?

- Como podemos modelar as classes "**Programador**" e "**Designer**"?

```java
public class Programador {

    private String nome;
    private String cpf;
    private double salario;
    private String linguagem;

    // Métodos GET e SET...
}
```

```java
public class Designer {

    private String nome;
    private String cpf;
    private double salario;
    private String softwareEdicao;

    // Métodos GET e SET...
}
```

Qual o problema?

- As classes **compartilham** boa parte de seus **atributos** e **métodos**.

- Problemas:

    1. Definir os **mesmos atributos e métodos** em todas as classes;
    2. **Se algo mudar** em uma classe, teremos que **mudar em todas as outras**.

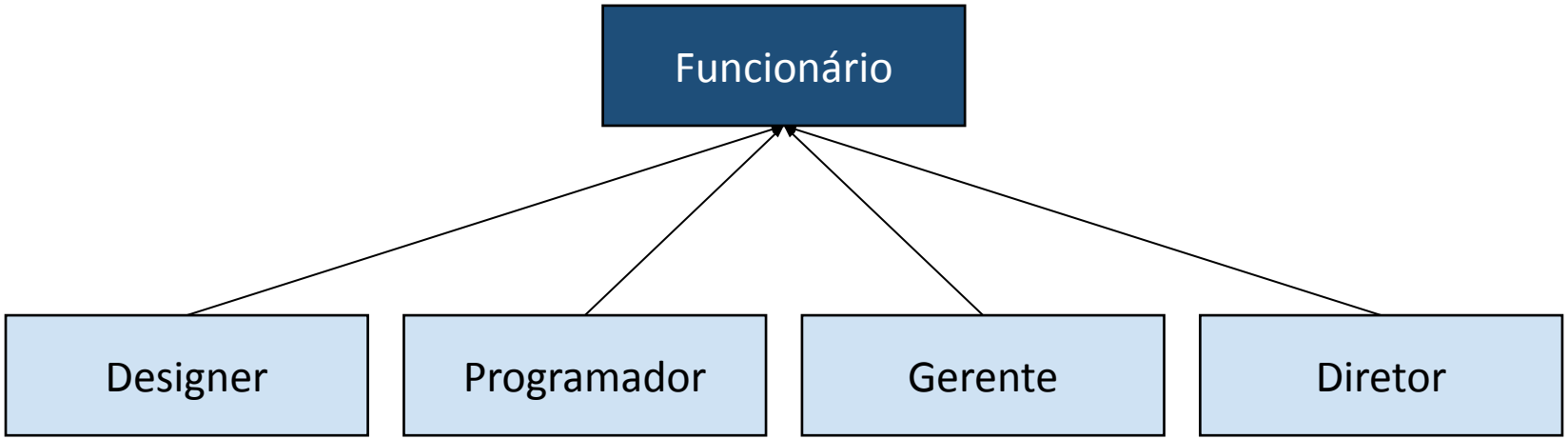
Solução!

- Podemos criar uma classe "**Funcionario**" e fazer com que as classes "**Programador**" e "**Designer**" herdem a classe "**Funcionario**".

- Com isso, os **atributos** e **métodos** de "Funcionario" farão parte de "Programador e "Designer".

- **Reutilizamos código** e evitamos o retrabalho de ter que definir os atributos e métodos comuns a todos os funcionários em cada classe.

Classes estruturadas de forma hierárquica



Herança em Java

```java
public class Funcionario {
    private String nome;
    private String cpf;
    private double salario;
    // Métodos GET e SET...
}
```

Superclasse

Subclasses

```java
public class Programador extends Funcionario
{
    private String linguagem;
    // Métodos GET e SET...
}
```

```java
public class Designer extends Funcionario
{
    private String softwareEdicao;
    // Métodos GET e SET...
}
```

Como saber que houve herança?

```
public class ClassePrincipal {
    public static void main(String[] args) {
        Programador programador = new Programador();

        programador.setNome("José");
        programador.setCpf("123456789-12");
        programador.setSalario(5000);
        programador.setLinguagem("Java");
    }
}
```

Métodos herdados da classe "Funcionario"

Nem sempre "herança" é a solução…

- Herança permite reusar código, **MAAAAAS....** somente quando faz sentido.
- Fazer uma classe herdar de outra só para reutilizar código é uma **péssima prática**.
- Só faz sentido usar herança se houver um **relacionamento** do tipo **É UM**, ou seja uma **especialização**:
  - Um "Programador" **É UM** "Funcionário" (então cabe usar herança);
  - Um "Programador" **NÃO É** um "Lutador" (apesar de ambos terem, nome, cpf e salário).

2.
Classes "final"

Herança

Classe "final"

- E se criarmos uma classe que **não queremos que seja herdada** por nenhum outra classe?
- Por exemplo, você pode ter uma classe que encapsula:
  - **Controle de um dispositivo de hardware**.
    - Faz uso de informação proprietária.
    - Não é desejável que os usuários da classe consigam sobrescrever esses métodos.
- Java permite que você **imponha essa restrição** com as classes final.

Classe "final"

```
public final class HardwareDriver {
    // Atributos

    void initialize() {
        // Código de inicialização
    }

    // Demais métodos
}
```

Garante que a classe não permita subclasses

3.
Modificadores de acesso

Herança

Herança e modificadores de acesso

- As **subclasses herdam todos os membros** (atributos e métodos) da **superclasse**;
- Mas, a **visibilidade dos atributos e métodos** depende de seus **modificadores de acesso**.

Modificadores de acesso

- **`private`**: somente objetos da própria classe enxergam.
- **`default`** (sem modificador): somente objetos da própria classe e das classes que estão no mesmo pacote enxergam.
- **`protected`**: somente objetos da própria classe, de suas subclasses e das classes que estão no mesmo pacote enxergam.
- **`public`**: objetos de qualquer classe enxergam.

Exemplo: uso do "public" **dentro** da classe

```
public class Funcionario {
	private double salario;
	// Métodos GET e SET...

	public void emitirSalario() {
		System.out.println(this.getSalario());
	}
}
```

```
public static void main(String[] args) {
	Funcionario func = new Funcionario("Gustavo", "123", 1000);
	func.emitirSalario();
}
```

Funciona! Executado dentro da mesma classe

Exemplo: uso do "public" **fora** da classe

```
public class Funcionario {
	private double salario;
	// Métodos GET e SET...

	public void emitirSalario() {
		System.out.println(this.getSalario());
	}
}
```

```
public class Programador extends Funcionario
{
	// Métodos GET e SET...
}
```

```
public static void main(String[] args) {
	Programador prog = new
Programador("Gustavo", "123", 2000);
	prog.emitirSalario();

}
```

Funciona! A visibilidade do método é pública

Exemplo: uso do "private" **fora** da classe

```
public class Funcionario {
	private double salario;
	// Métodos GET e SET...

	private void emitirSalario() {
		System.out.println(this.getSalario());
	}
}
```

```
public class Programador extends Funcionario
{
	// Métodos GET e SET...
}
```

```
public static void main(String[] args) {
	Programador prog = new
Programador("Gustavo", "123", 2000);
	prog.emitirSalario();

}
```

Não funciona! Executado fora da classe definida

Encapsulamento

- Encapsulamento é um conceito de programação orientada a objetos que **liga os dados e funções**.
- O encapsulamento mantém ambos **seguros de interferência** externa e **má utilização**.
- O encapsulamento de dados leva ao importante conceito de POO de **ocultação de dados**, tornando as informações **privadas apenas a quem as possui**.

Exemplo: uso do "protected" **dentro** da classe

```
public class Funcionario {
	private double salario;
	// Métodos GET e SET...

	protected void emitirSalario() {
		System.out.println(this.getSalario());
	}
}
```

```
public static void main(String[] args) {
    Funcionario func = new Funcionario("Gustavo", "123", 1000);
    func.emitirSalario();
}
```

Funciona! Executado dentro da mesma classe

Exemplo: uso do "**protected**" fora da classe

```
public class Funcionario {
	private double salario;
	// Métodos GET e SET...

	protected void emitirSalario() {
		System.out.println(this.getSalario());
	}
}
```

```
public class Programador extends Funcionario
{
	// Métodos GET e SET...
}
```

```
public static void main(String[] args) {
	Programador prog = new
Programador("Gustavo", "123", 2000);
	prog.emitirSalario();

}
```

Funciona! A superclasse protegeu para as subclasses

	Private Member	Default Member	Protected Member	Public Member
Visible within same class	Yes	Yes	Yes	Yes
Visible within same package by subclass	No	Yes	Yes	Yes
Visible within same package by non-subclass	No	Yes	Yes	Yes
Visible within different package by subclass	No	No	Yes	Yes
Visible within different package by non-subclass	No	No	No	Yes

Fonte: Herbert Schildt. Java™ A Beginner's Guide. Ninth Edition, McGraw Hill, 2022.

Modificadores de acesso mais utilizados

- **Métodos** em java costumam ser **public**, a menos que só faça sentido utilizá-los por objetos da própria classe. Nesses casos, usamos **private**.

- **Atributos** em Java costumam ser **private**, muito raramente são **protected** e quase nunca são **public** ou **default**.

- O modificador **default** (ausência de modificador) é o que temos usado até agora, mas praticamente **não é usado na prática**.

4.
Classes abstratas

Herança

O que são classes abstratas?

- São classes que **possuem métodos abstrato**.
  - **Transfere** a definição dos métodos abstratos para as subclasses;
  - Possibilita o **polimorfismo**.
- Classes abstratas **não podem ser instanciadas**, somente suas subclasses.
- Subclasses de uma classe abstrata **que não implementarem os métodos abstratos** terão **erro de compilação.**
- Classes abstratas podem ter **métodos concretos** e **abstratos**.

Exemplo: classe abstrata

Classe abstrata

```
public abstract class Funcionario {
    private String nome;
    private String cpf;
    private double salario;

    // Métodos GET e SET...

      public abstract double calcularBonificacao();
}
```

Cada subclasse deve definir a sua forma de implementar o **método abstrato**

Comportamentos específicos

E se todo **funcionário** tivesse direito a uma **bonificação no fim do ano**?

Porém, cada tipo de funcionário tem uma **lógica de cálculo de bonificação diferente** na empresa…

Exemplo: sobrescrevendo um método da superclasse

```java
public class Programador extends Funcionario
{

    private String linguagem;

    // Métodos GET e SET...

    public double calcularBonificacao() {
        double bonificacao = salario * 0.2;
        return bonificacao;
    }
}
```

```java
public class Designer extends Funcionario {

    private String softwareEdicao;

    // Métodos GET e SET...

    public double calcularBonificacao() {
        double bonificacao = salario *
0.1;
        return bonificacao;
    }
}
```

Implementação **específica** do método abstrato para cada subclasse

Observando o funcionamento dos métodos

```
public class ClassePrincipal {

    public static void main(String[] args) {
        Programador programador = new Programador();
        programador.setNome("José");
        programador.setSalario(5000);

        Designer designer = new Designer();
        designer.setNome("Maria");
        designer.setSalario(5000);

        System.out.println(programador.calcularBonificacao());
        System.out.println(designer.calcularBonificacao());
    }
}
```

```
1000
500
```

5. Polimorfismo

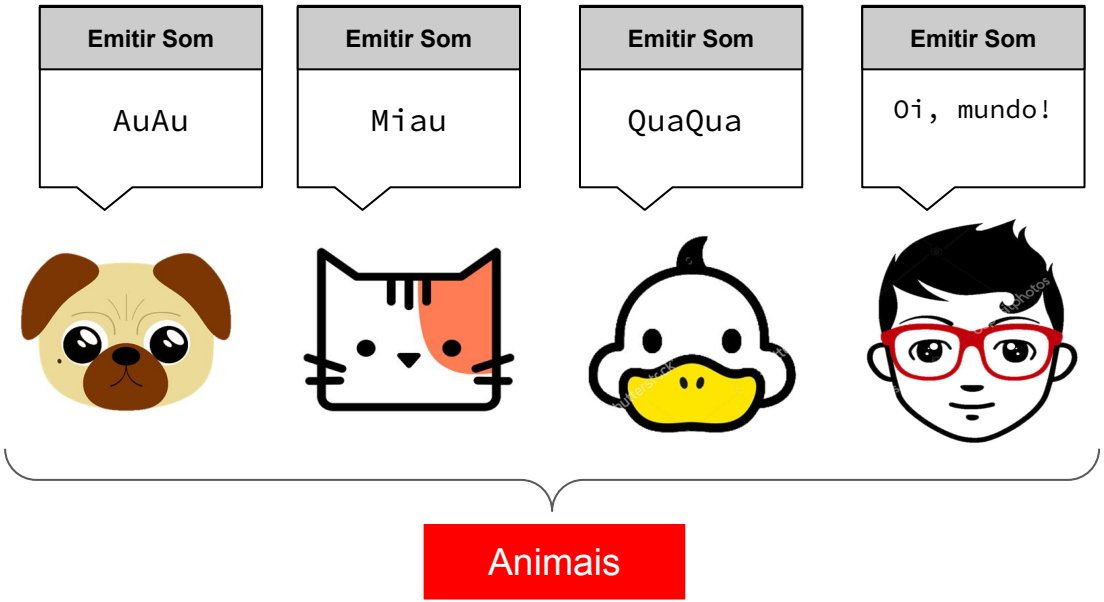
Herança

O que é polimorfismo?

- **Polimorfismo** é o princípio pela qual **duas ou mais subclasses** podem **invocar métodos que têm a mesma assinatura**, mas **comportamentos distintos** baseados na sua **especialidade**.

- Ele é usado na intenção de **manter métodos com a mesma assinatura**. Isso ajuda a não criar muitos métodos diferentes.

Exemplo: polimorfismo



E se precisarmos de uma implementação padrão para um método?

- **Não precisa**, necessariamente, de uma **classe abstrata**.
- Você pode **sobrescrever o método da superclasse** na subclasse.
- Quando alguém chama um **método sobrescrito**, a versão que será chamada é a da **subclasse**.

Sobrescrita de métodos

- Cuidado para não confundir **sobrescrita** com **sobrecarga** de método.

- Caso você **não queira que uma subclasse sobrescreva um dos métodos** da classe que você criou, basta fazer com que o método seja "**final**".

Exemplo: sobrescrita de métodos

```
public class Funcionario {
    // Atributos
    public double calcularBonificacao() {
        double bonificacao = salario * 0.1;
        return bonificacao;
    }
}
```

```
public class Programador extends Funcionario
{
    private String linguagem;
    // Métodos GET e SET...
    public double calcularBonificacao() {
      double bonificacao = salario * 0.2;
      return bonificacao;
    }
}
```

```
public class Designer extends Funcionario
{
    private String softwareEdicao;
    // Métodos GET e SET...
}
```

Exemplo: sobrescrita de métodos

```
public class Funcionario {
    // Atributos
    public double calcularBonificacao() {
        double bonificacao = salario * 0.1;
        return bonificacao;
    }
}
```

A classe não precisa ser abstrata

implementação padrão

```
public class Programador extends Funcionario
{
    private String linguagem;
    // Métodos GET e SET...
    public double calcularBonificacao() {
      double bonificacao = salario * 0.2;
      retur
    }
}
```

Este método sobrescreve o da "superclasse"

```
public class Designer extends Funcionario
{
    private String softwareEdicao;
    // Métodos GET e SET...
}
```

Usou a implementação padrão da superclasse "Funcionário"

Padronização de código

```
public static void main(String[] args) {
    Funcionario func = new Funcionario("José", "334", 5000);
    Programador prog = new Programador("Gustavo", "123", 2000);
    Designer des = new Designer("Maria", "334", 10000);

    Funcionario[] lista_func = {prog, func, des};

    for (int i = 0; i < lista_func.length; i++) {
        lista_func[i].emitirSalario();
    }
}
```

Padronização de código

Regras de sobrescrita de métodos

- A **lista de parâmetros** do método da **subclasse** precisa ter o mesmo **número**, **sequência** e **tipos** de parâmetros do método que será **sobrescrito da superclasse**.

- O **modificador de acesso** do método não pode ser mais restritivo. Por exemplo, um método **private** não pode sobrescrever um método **public**, mas o contrário é possível.

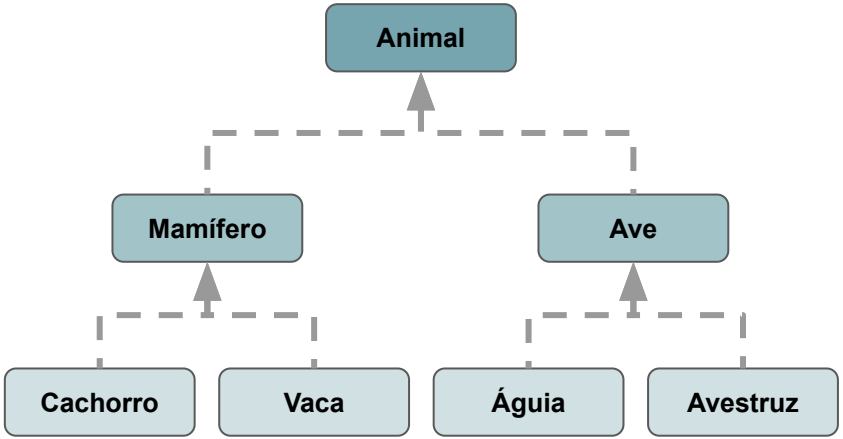
E se a subclasse quiser chamar o método da superclasse?

```
public class Gerente extends Funcionario {
    // Atributos...
    // Métodos GET e SET...
    public double calcularBonificacao() {
        double bonificacao = super.calcularBonificacao();
        double bonificacao += 1000;
        return bonificacao;
   }
}
```

Resumo e benefícios | Herança e polimorfismo

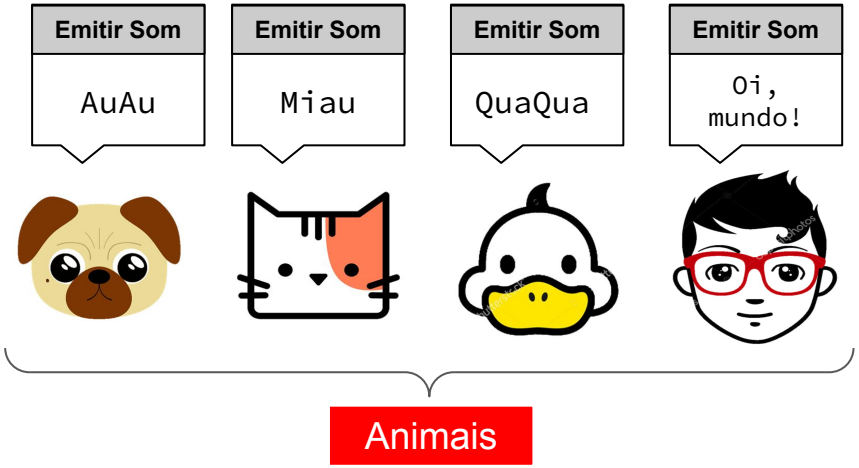
Herança

* Reaproveita linhas de código;
* Economiza tempo no desenvolvimento.



Polimorfismo

* Mantém os métodos com a mesma assinatura;
* Otimiza a chamada de métodos para diferentes classes.



6. Considerações finais

Herança

O que aprendemos hoje?

- O conceito de herança e sua aplicação na programação orientada a objetos;
- O significado e o uso da palavra-chave "final" em relação a classes, métodos e atributos;
- Os modificadores de acesso (public, private, protected) e seu papel na encapsulação;
- O conceito de classes abstratas e seu propósito na modelagem de classes;
- O conceito de polimorfismo e sua importância na programação orientada a objetos.

Próxima aula…

-

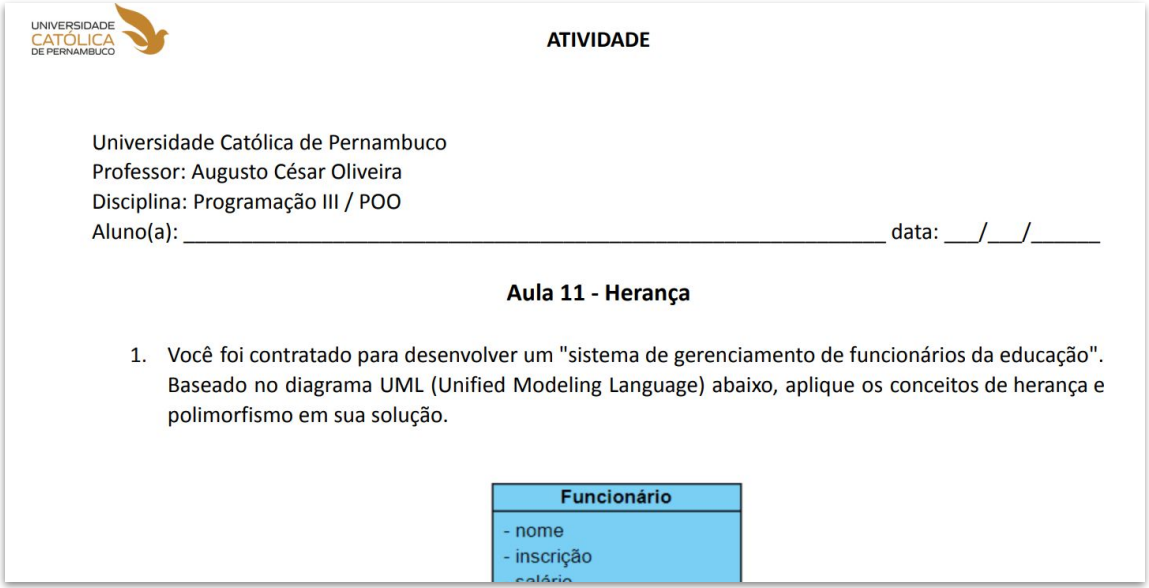
7.
Exercício de fixação

Teams

Exercício de fixação

Herança

- **Link da atividade**: clique aqui.



UNIVERSIDADE CATÓLICA DE PERNAMBUCO

ATIVIDADE

Universidade Católica de Pernambuco
Professor: Augusto César Oliveira
Disciplina: Programação III / POO
Aluno(a): _____ data: ___/___/_____

Aula 11 - Herança

1. Você foi contratado para desenvolver um "sistema de gerenciamento de funcionários da educação". Baseado no diagrama UML (Unified Modeling Language) abaixo, aplique os conceitos de herança e polimorfismo em sua solução.

Funcionário
- nome - inscrição - salário

Aula 10

Herança

Programação III

Prof. Augusto César Oliveira
augusto.oliveira@unicap.br