- **UESC - Universidade Estadual de Santa Cruz**

Cap 3 – Herança e Polimorfismo

Parte 1 - Herança

Disciplina: Linguagem de Programação III
Professor: Otacílio José Pereira

Plano de Aula

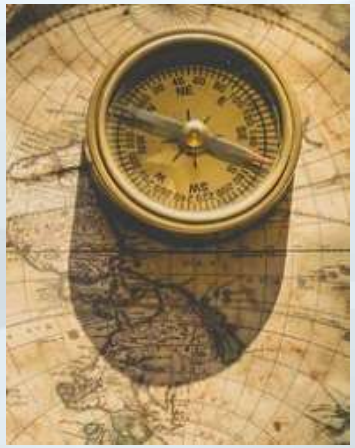
- ***Objetivos***
  - Compreender um dos princípios mais importantes da Orientação a Objetos, o princípio e mecanismo de herança
- ***Tópicos***
  - Contexto: Tópicos já explorados
  - Herança: Introdução
  - Herança: Codificando
    - Extends
    - Sobrecarregando os métodos
  - Outros mecanismos
    - Modificador de Acesso (Protected)
    - Super
    - @Override
  - Exercícios

Contexto

- Onde estamos?
- Foco agora!
- Cenário de exemplo

Onde estamos?

- *Considerando nosso planejamento inicial*

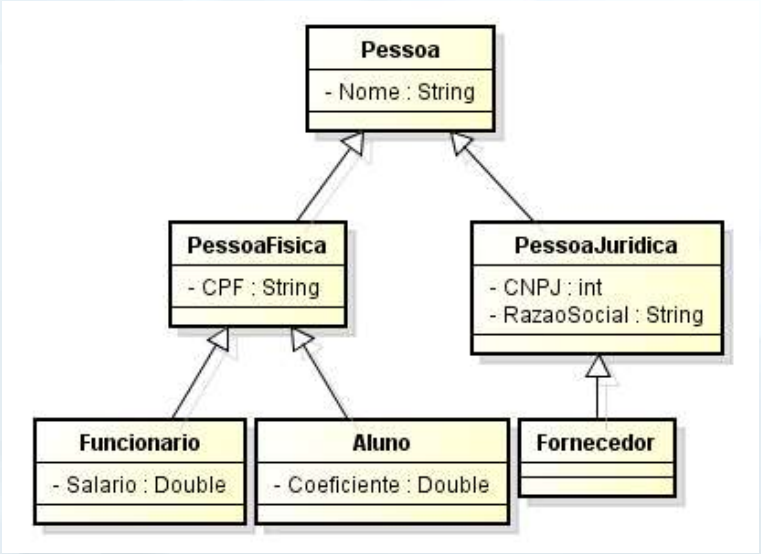


- Capítulo 1 – Introdução
- Capítulo 2 - Conceitos básicos de Orientação a Objetos
  - OO, classes e objetos
  - Atributos, métodos, encapsulamento, getters, setters
  - Outros tópicos: Atributos de classe (static), tipos primitivos, tipos por referência

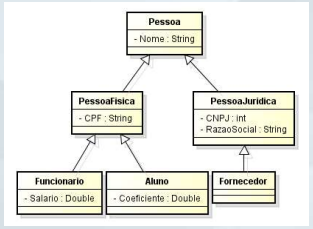
Foco

- Capítulo 3 – Herança e Polimorfismo
- Capítulo 4 – Classes abstratas e interfaces
- Capítulo 5 – Generics, collections e outros tópicos
- Capítulo 6 – Desenvolvimento de um projeto em OO

Foco agora



- Compreender principalmente os conceitos de herança
- Juntamente com como implementá-la na prática
- Discutindo seus benefícios para o desenvolvimento de sistemas

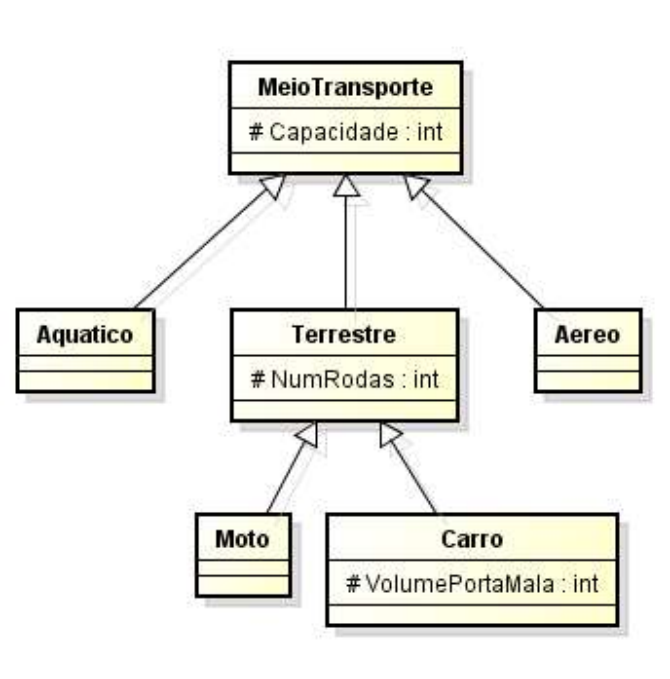



```
public class PessoaFisica extends Pessoa{

    String CPF;

    PessoaFisica()
    {
        CPF = "00000000000";
    }
```

Cenário / Problema

Herança na Orientação a Objetos

Software com bom projeto, com e para reuso

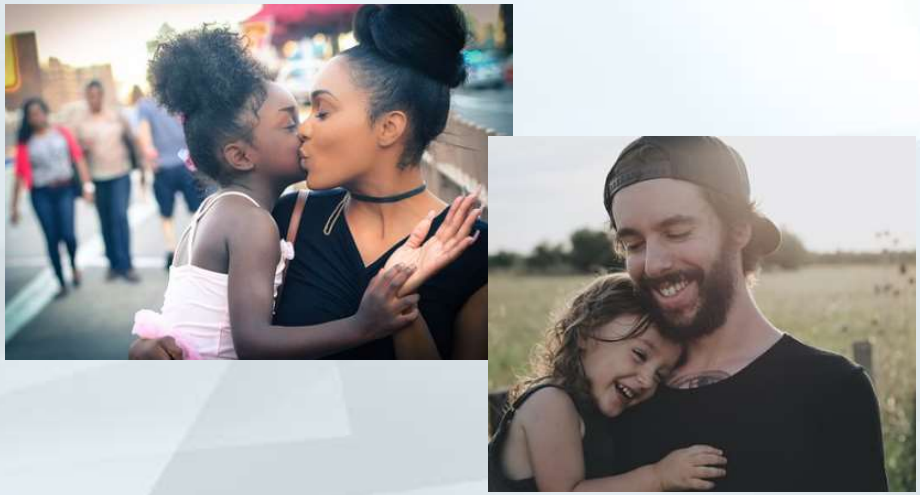


Introdução à Herança

- Introdução abrangente
- Definição
- Conceitos
- Cenários

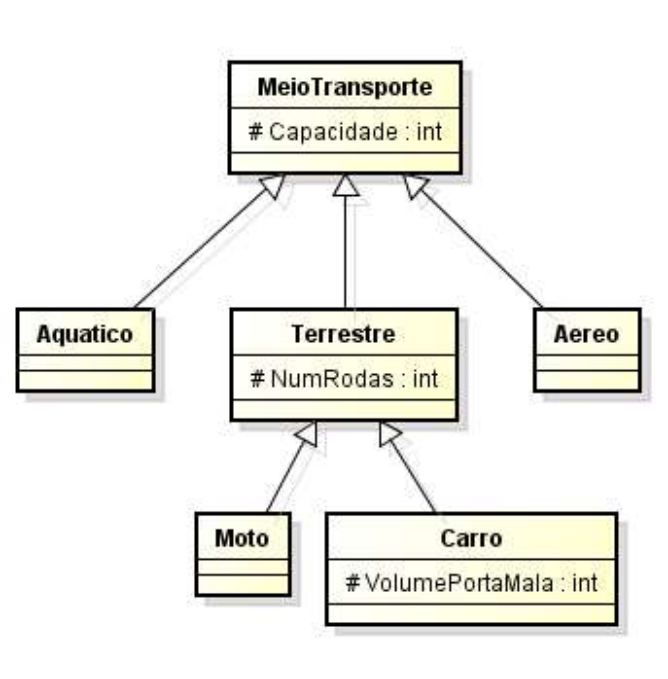
Herança

- O que herdamos de nossos pais?



- Herança
  - Permite que classes herdem características de classe pai/mãe
  - É um dos princípios mais importantes da OO
  - Por meio da herança, classes filhas herdam atributos e métodos das classes superiores
  - E daí consegue-se um bom recurso de reuso e de qualidade de software

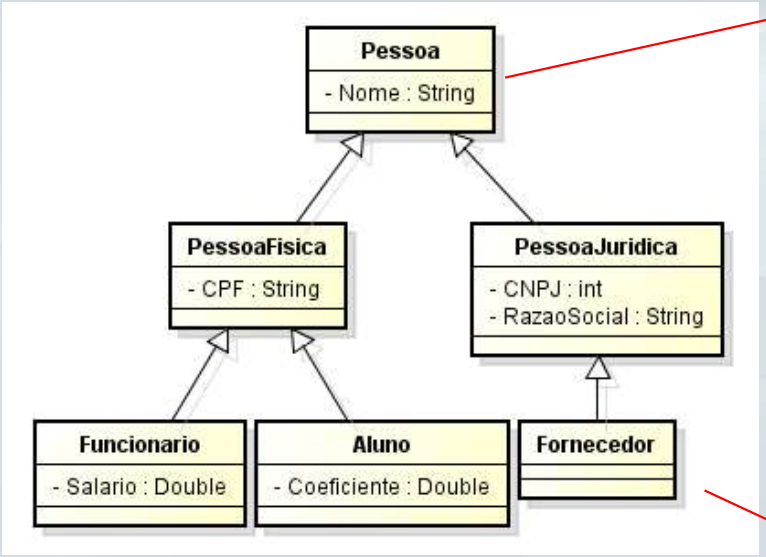
Herança



- Conforme Deitel
  - "É uma forma de reutilização de software em que uma nova classe é criada absorvendo características de uma classe existente e aprimorando novas capacidades ou modificações."
  - Com a herança você pode economizar tempo durante o desenvolvimento de um programa baseando novas classes no software já existente e testado, depurado e de alta qualidade.

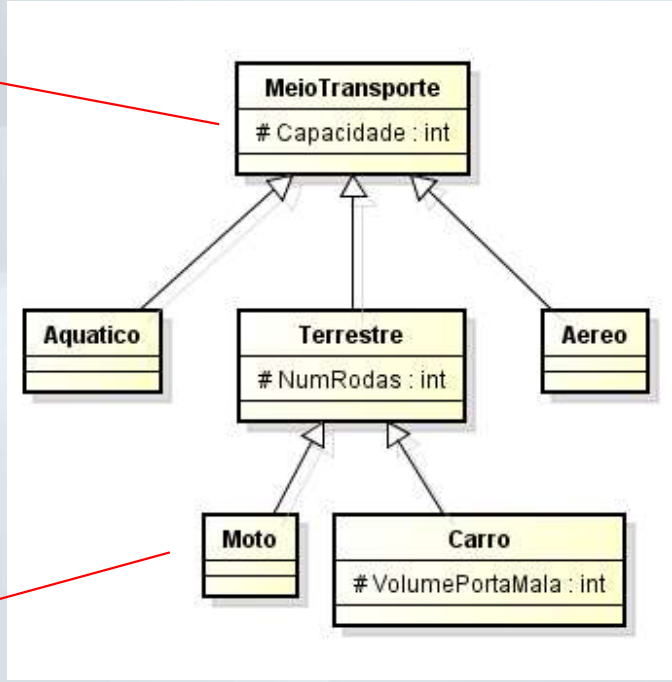
Herança

- Alguns conceitos
  - Dois exemplos permitem entender alguns conceitos relacionados a herança



Superclasse ou
Classe Básica

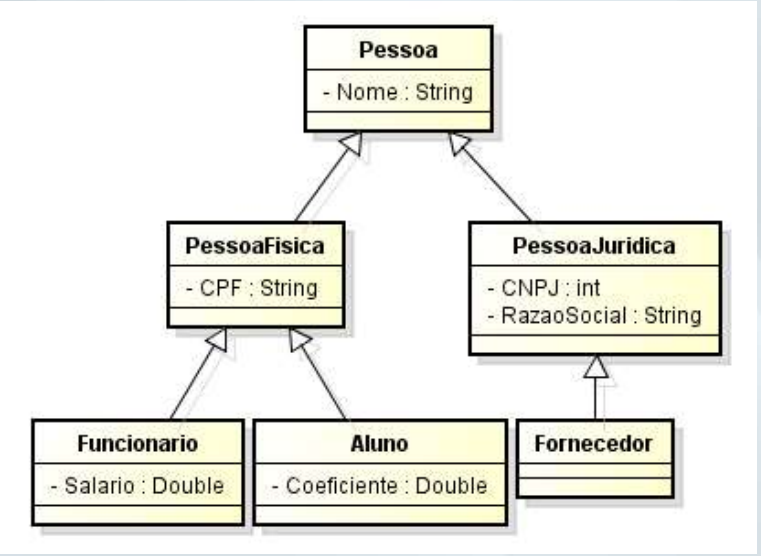
Subclasse ou
Classe Derivada



Herança

- Uma classe derivada herda os atributos e métodos da superclasse
  - **PessoaFisica** (subclasse) herdou o atributo **Nome** de **Pessoa** (superclasse)
  - **Funcionario** (subclasse) herdou ao atributo **CPF** de **PessoaFisica** (Superclasse)

- Observe:
  - A classe PessoaFísica precisou redefinir o atributo Nome?
  - A classe Funcionário precisou redefinir o atributo Salario?
  - O que a classe Aluno tem de específico em relação às outras?



Herança

- Neste momento, busque analisar outros cenários. Em cada um deles identifique
  - Quais as superclasses e as classes derivadas?
  - Quais semelhanças e diferenças entre as classes?
  - Quais atributos e métodos seriam alocados nas superclasses e nas classes filhas?
  - Quais outros cenários de exemplo de herança?
    - Pessoa, Aluno, Professor e Administrativo
    - Curso, Curso Presencial, Curso EAD
    - FormaGeometrica, Quadrado, Triângulo, Círculo

```java
public class Pessoa {

    String Nome;

    Pessoa()
    {
        Nome = "Novo Nome";
    }
```

```java
public class PessoaFisica extends Pessoa{

    String CPF;

    PessoaFisica()
    {
        CPF = "00000000000";
    }
```

Herança (Programando)

- Uso “extends”
- Especializando métodos
- Discussões gerais

Herança

- Para implementar a herança utiliza-se o “extends”

```java
public class Pessoa {

    String Nome;

    Pessoa()
    {
        Nome = "Novo Nome";
    }

    public String getNome() {
        return Nome;
    }

    public void setNome(String Nome) {
        this.Nome = Nome;
    }

}
```

```java
public class PessoaFisica extends Pessoa{

    String CPF;

    PessoaFisica()
    {
        CPF = "00000000000";
    }

    public String getCPF() {
        return CPF;
    }

    public void setCPF(String CPF) {
        this.CPF = CPF;
    }

}
```

  - Observe a classe PessoaFisica: Quais atributos e métodos foram reaproveitados de Pessoa?

Herança

- Discussões para no nosso caso
- Em um banco ou Fintech, quais os tipos de contas podem existir?
    - Quais tipos de contas podemos ter em um banco?
    - O que há de comum nos tipos de contas?
    - O que há de específico em cada tipo de conta?

- Exercício 2
  - Crie a classe ContaCorrente que herda de Conta
  - Adicione o atributo Limite (Double)
  - Encapsule o atributo (use o private e os métodos get e set)
  - Sobreescreva o método Sacar que permite sacar juntamente com o limite
  - Teste a classe na classe principal do projeto Banco

Herança

- Códigos de exemplo

```
public class ContaCorrente extends Conta{

    private double Limite = 200;

    public double getLimite() {
        return Limite;
    }

    public void setLimite(double Limite) {
        this.Limite = Limite;
    }

    @Override
    void Sacar(Double pValor)
    {
        if (pValor < (Saldo+Limite) )
            this.Saldo = this.Saldo - pValor;
        else
            System.out.println(" Não foi possível sacar. ");
    }

}
```

```
public class Banco_v5 {
    public static void main(String[] args) {
        ContaCorrente c1 = new ContaCorrente();

        Scanner s = new Scanner( System.in );

        System.out.printf(" Entre com o nome do titular da conta : ");
        c1.setNomeTitular(s.nextLine());
        System.out.printf(" Entre com a agência da conta         : ");
        c1.setAgencia(s.nextInt());
        System.out.printf(" Entre com o numero da conta          : ");
        c1.setNumeroConta(s.nextInt());
        c1.Imprimir();

        System.out.println(" Qual operação realizar ?");
        System.out.println("    (1) Saque ");
        System.out.println("    (2) Depósito ");
        System.out.print("  Operação: ");
        int operacao = s.nextInt();

        System.out.print(" Qual valor ? ");
        double valor = s.nextDouble();
        if (operacao == 1)
            c1.Sacar(valor);
        else
            c1.Depositar(valor);

        c1.Imprimir();
    }
}
```

- Perceba que agora o objeto é da Classe ContaCorrente
- Ele utiliza setAgencia, setNumero da Classe Conta
- E quanto invocar Sacar, o método que considera limite que será usado

Herança (Refinando)

- Modificador Protected
- @Override
- Super

```
ContaCorrente()
{
    super();
    Limite = 200;
}

ContaCorrente(int pAgencia, int pNumeroConta,
              String pNomeTitular, double pSaldo, double pLimite)
{
    super(pAgencia, pNumeroConta, pNomeTitular, pSaldo);
    this.Limite = pLimite;
}
```

Modificador de acesso “protected”

- Ao estudar encapsulamento, inicialmente usam-se os modificadores “public” e “private”
  - public : o atributo ou método tem acesso fora da classe, em qualquer parte do sistema
  - private : o atributo ou método tem acesso apenas na classe em que foi definido
- O modificador “protected” permite que o atributo ou método seja acessado, além da classe onde é definido, também nas classes da hierarquia
- Por exemplo, se for necessário acessar Agencia, NumeroConta e outros atributos de Conta em ContaCorrente (subclasse na hierarquia) estes devem ser definidos com proteced

```
public class Conta {

    protected int       Agencia;
    protected int       NumeroConta;
    protected String    NomeTitular;
    protected double  Saldo = 0;
```

Anotação @override

- Nas subclasses, quando precisamos de especializar os métodos é importante utilizar a anotação @override
- Por exemplo, nos métodos imprimir ou sacar
- Neste caso você está indicando para o compilador que o método é uma sobrecarga (override) do método da superclasse
- Ele portanto faz a checagem se você escreveu corretamente o cabeçalho do método. É uma forma de garantir a consistência do seu projeto e programa.

```
3   public class ContaCorrente extends Conta{
4
5       double  Limite;
6
7       @Override
@       public void Sacar(Double pValor)
9       {
10          if (pValor < ( this.Valor + this.Limite ))
11              this.Valor = this.Valor - pValor;
12      }
13  }
```

Super

- Ao se escrever um método da subclasse pode ser que um método da superclasse seja invocado. Neste caso é usado o termo “super”
  - Por exemplo, o método imprimir de ContaCorrente pode invocar o “Imprimir” de conta e complementar a impressão com os dados de ContaCorrente

```
@Override
void Imprimir()
{
    super.Imprimir();
    System.out.println(" Limite              : " + this.Limite);
    System.out.println(" ------------------------------------ ");
}
```

Super

- Exercício 5
  - Crie o método de imprimir que imprime os dados da classe Conta juntamente com os dados da classe ContaCorrente
- Exercício 6
  - Altere o método Imprimir para que a parte de classe Conta (superclasse) seja realizada por meio da palavra super
- Exercício 7
  - Crie os construtores padrão e com parâmetros para a classe ContaCorrente e na implementação deles acrescente a chamada ao construtor da superclasse através de “super();”

Super

- Exemplo 1 : Construtor de ContaCorrente que recebe limite.

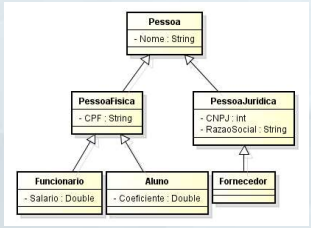
```
ContaCorrente()
{
    super();
    Limite = 200;
}

ContaCorrente(int pAgencia, int pNumeroConta,
              String pNomeTitular, double pSaldo, double pLimite)
{
    super(pAgencia, pNumeroConta, pNomeTitular, pSaldo);
    this.Limite = pLimite;
}
```

- Exemplo 2 : Imprimir que executa o imprimir da superclasse

```
@Override
void Imprimir()
{
    super.Imprimir();
    System.out.println(" Limite            : " + this.Limite);
    System.out.println(" ---------------------------------------- ");
}
```

Reflexões

- Representação do cenário/problema
- Reaproveitamento
- Frameworks



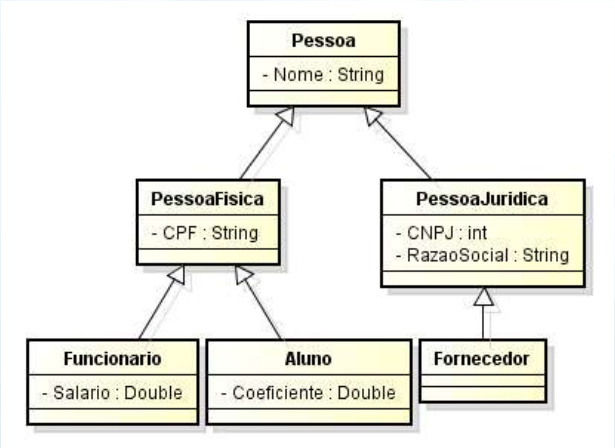

```
public class PessoaFisica extends Pessoa{

    String CPF;

    PessoaFisica()
    {
        CPF = "00000000000";
    }
```

Cenário / Problema

Herança na Orientação a Objetos

Software com bom projeto, com e para reuso

Reflexões



- Pense a respeito da aula de hoje, foi realizado muito código?
- É possível perceber que muito do que usamos foi reaproveitado, reusado.
- Apenas fizemos incrementos pontuais do que precisávamos
- Isso porque usar Herança permite aproveitar bastante, herdar características e códigos já criados.

Conclusões

- Retomando Plano de Aula
- Revisão
- Para saber mais

Conquistou o exercício?

- ***Objetivos***
  - Exercitar os conceitos iniciais de Orientação a Objetos

- ***Tópicos***
  - Visão geral da Introdução à Orientação a Objetos
  - Classes e objetos
  - Declaração, instanciação e inicialização
  - Atributos, Métodos e Estado
  - Encapsulamento
  - Modificadores de acesso
  - Métodos getters e setters
  - Construtores

- **UESC - Universidade Estadual de Santa Cruz**

Cap 2 – Introdução à Orientação a Objetos

Exercício sobre a Introdução à Orientação a Objetos

Disciplina: Linguagem de Programação III
Professor: Otacílio José Pereira