

# Herança

## POO

### Prof. Marcio Delamaro

# Herança - dicionário

- s.f. Bem, direito ou obrigação transmitidos por disposição testamentária ou por via de sucessão.
- Legado, patrimônio.
- Fig. Condição, sorte, situação que se recebe dos pais.
- Genét. Conjunto de caracteres hereditários transmitidos pelos genes; hereditariedade.

# Herança – wikipedia

- In object-oriented programming, inheritance is when an object or class is based on another object (prototypal inheritance) or class (class-based inheritance), using the same implementation (inheriting from an object or class) specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior). It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces. The relationships of objects or classes through inheritance give rise to a hierarchy. Inheritance was invented in 1967 for Simula.

# A ideia...

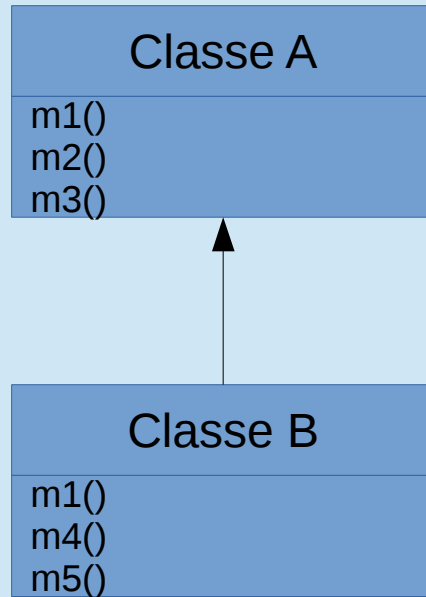
- ...é criar uma classe baseada em outra já existente
- A classe “filha” pode ampliar
- Ou modificar a classe já existente
- E isso pode ser repetir para a classe “filha”.

# Quer que desenhe?

Classe A
m1()
m2()
m3()

```
A varA = new A();
varA.m1();
varA.m2();
varA.m3();
```

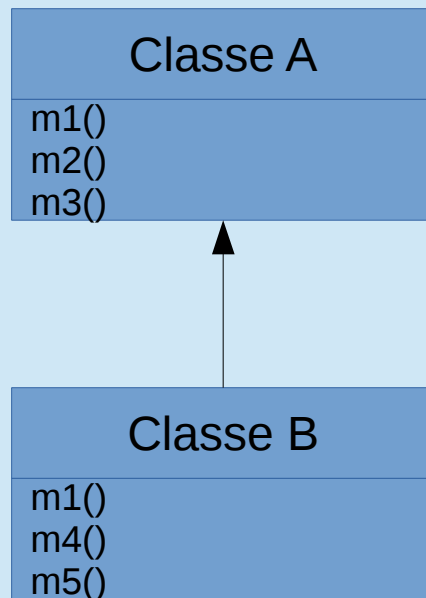
# Quer que desenhe?



```

A varA = new A();
varA.m1();
varA.m2();
varA.m3();
  
```

# Quer que desenhe?

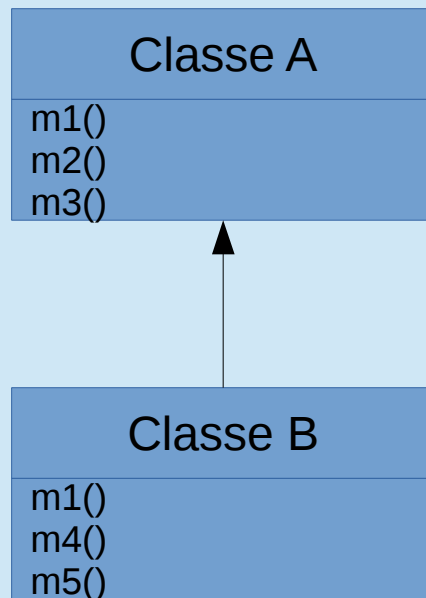
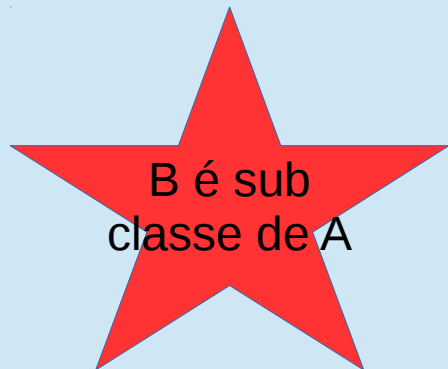


```

A varA = new A();
varA.m1();
varA.m2();
varA.m3();
    
```



# Quer que desenhe?



```

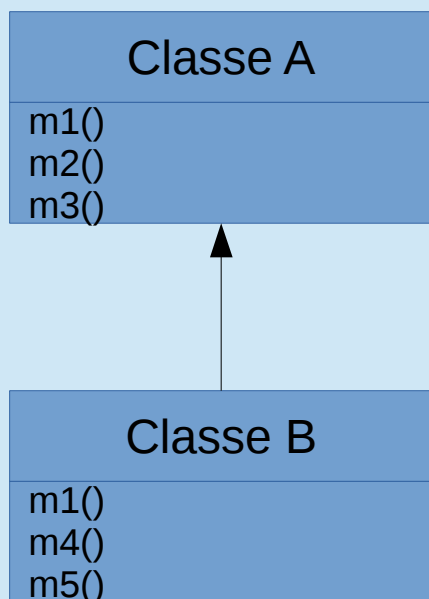
A varA = new A();
varA.m1();
varA.m2();
varA.m3();
    
```





# Quer que desenhe?

B herda de A  
B é filha A é mãe  
A é superclasse  
B é subclasse  
B estende A



```

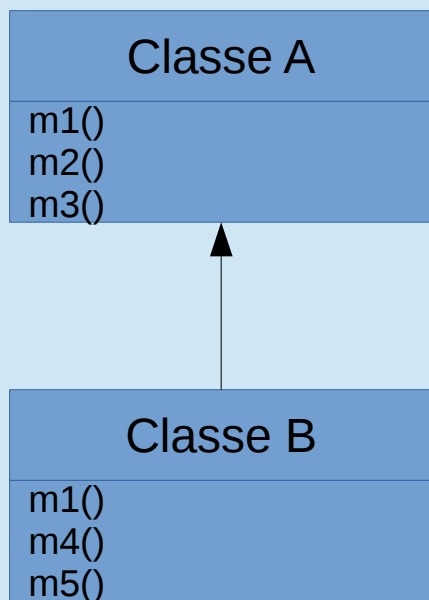
A varA = new A();
varA.m1();
varA.m2();
varA.m3();
    
```

B estende A

B é sub  
classe de A

# Quer que desenhe?

B herda de A  
B é filha A é mãe  
A é superclasse  
B é subclasse  
B estende A



```

A varA = new A();
varA.m1();
varA.m2();
varA.m3();
  
```

```

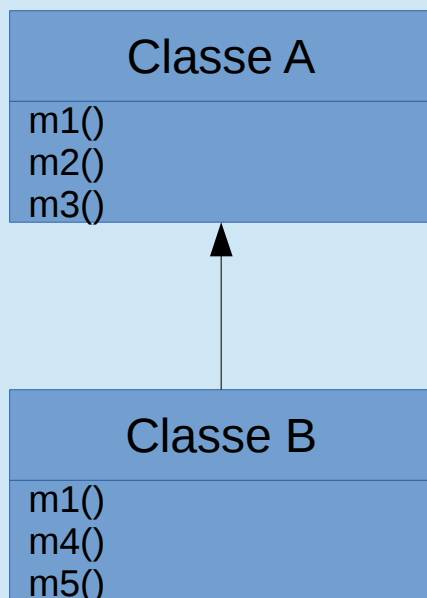
B varB = new B();
varB.m1();
varB.m4();
varB.m5();
  
```

B estende A

B é sub  
classe de A

# Quer que desenhe?

B herda de A  
B é filha A é mãe  
A é superclasse  
B é subclasse  
B estende A



```
A varA = new A();
varA.m1();
varA.m2();
varA.m3();
```

```
B varB = new B();
varB.m1();
varB.m4();
varB.m5();
varB.m2();
varB.m3();
```

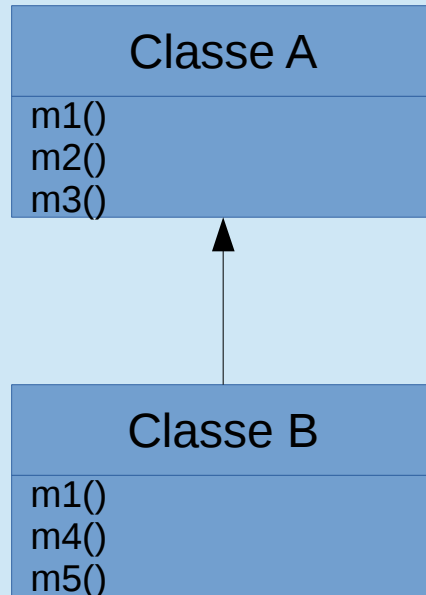
B estende A

B é sub  
classe de A

Os elementos de A fazem parte de B

# Quer que desenhe?

B herda de A  
B é filha A é mãe  
A é superclasse  
B é subclasse  
B estende A



```

A varA = new A();
varA.m1();
varA.m2();
varA.m3();
  
```

```

B varB = new B();
varB.m1();
varB.m4();
varB.m5();
varB.m2();
varB.m3();
  
```

Os elementos de B  
ampliam o conteúdo de A

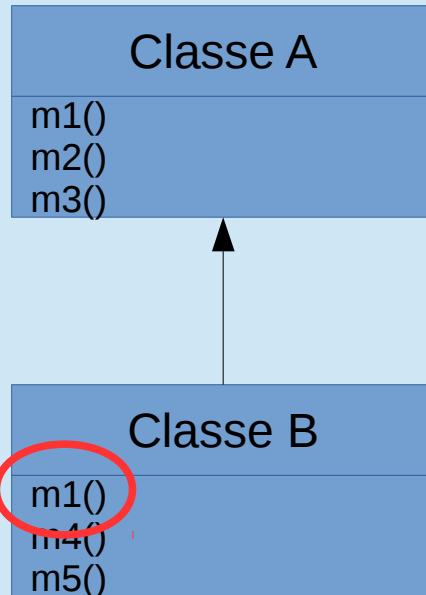
Os elementos de A fazem parte de B

B estende A

B é sub  
classe de A

# Quer que desenhe?

B herda de A  
B é filha A é mãe  
A é superclasse  
B é subclasse  
B estende A



```

A varA = new A();
varA.m1();
varA.m2();
varA.m3();
  
```

```

B varB = new B();
varB.m1();
varB.m4();
varB.m5();
varB.m2();
varB.m3();
  
```

Elemento de B altera elemento de A

Os elementos de B ampliam o conteúdo de A

B estende A

B é sub  
classe de A

Os elementos de A fazem parte de B

# Exemplo trivial

- Classe Pessoa: idade, altura, peso
- *getters e setters*
- computaIMC: computa índice de massa corporal

# Classe Pessoa – atributos

```
public class Pessoa {  
    /** peso da pessoa em kg */  
    private double peso;  
  
    /** idade da pessoas (anos completos) */  
    private int idade;  
  
    /** altura da pessoa em metro */  
    private double altura;
```

# Classe pessoa – construtor

```
public Pessoa(double peso, double altura, int idade)
{
    this.peso = peso;
    this.altura = altura;
    this.idade = idade;
}
```



# Classe Pessoa – métodos

```
public double getAltura() {  
    return altura;  
}
```

```
public void setAltura(double altura) {  
    this.altura = altura;  
}
```

```
public double getIMC() {  
    return getPeso() / (getAltura() * getAltura());  
}
```

# Obesidade

- O indicativo de obesidade é variado para homens e mulheres
- Por isso vamos diferenciar objetos
- Vamos criar duas classes
  - PessoaHomem
  - PessoaMulher

# Classe PessoaMulher

- Possui os mesmos atributos de Pessoa
- Portanto, deve ter os mesmos getters e setters
- O IMC também precisa ser computado
- E existe uma regra para classificar a mulher por nível de obesidade

Condição	IMC em Mulheres	IMC em Homens
abaixo do peso	< 19,1	< 20,7
no peso normal	19,1 - 25,8	20,7 - 26,4
marginalmente acima do peso	25,8 - 27,3	26,4 - 27,8
acima do peso ideal	27,3 - 31,1	27,8 - 32,3
obeso	> 31,1	> 32,3

***IMC = peso / altura<sup>2</sup> (peso em kg e altura em m)***



# Classe PessoaMulher

```
public class PessoaMulher extends Pessoa {
```

```
PessoaMulher pm = new PessoaMulher(51.3, 1.58, 35);  
double x = pm.getIMC();
```

# Classe PessoaMulher

```
public class PessoaMulher extends Pessoa {
```

```
PessoaMulher pm = new PessoaMulher(51.3, 1.58, 35);  
double x = pm.getIMC();
```

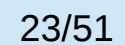
## O que tem de errado aqui?



}

## O que tem de errado aqui?

## Falta um construtor



# Classe PessoaMulher – construtor

```
public class PessoaMulher extends Pessoa {  
    public PessoaMulher(double peso, double altura,  
                        int idade)  
  
    {  
  
        super(peso, altura, idade);  
  
    }  
  
}
```



# Classe PessoaMulher – método

```
public class PessoaMulher extends Pessoa {  
    public String classificaObesidade() {  
        double imc = getIMC();  
        if (imc < 19.1) return "Abaixo do peso";  
        if (imc < 25.8) return "Peso normal";  
        if (imc < 27.3) return "Marginalmente acima do peso";  
        if (imc < 31.1) return "Acima do peso";  
        return "Obeso";  
    }  
}
```

# Classe PessoaMulher – método

```
public class PessoaMulher extends Pessoa {  
    public String classificaObesidade() {  
        double imc = this.getIMC();  
        if (imc < 19.1) return "Abaixo do peso";  
        if (imc < 25.8) return "Peso normal";  
        if (imc < 27.3) return "Marginalmente acima do peso";  
        if (imc < 31.1) return "Acima do peso";  
        return "Obeso";  
    }  
}
```

# Classe PessoaMulher – método

```
public class PessoaMulher extends Pessoa {
    public String classificaObesidade() {
        double imc = super.getIMC();
        if (imc < 19.1) return "Abaixo do peso";
        if (imc < 25.8) return "Peso normal";
        if (imc < 27.3) return "Marginalmente acima do peso";
        if (imc < 31.1) return "Acima do peso";
        return "Obeso";
    }
}
```

Vamos reservar o uso de super para quando houver ambiguidade entre a classe atual e a superclasse.

# Usando as classes

```
Pessoa p = new Pessoa(72.5, 1, 73, 52);  
PessoaMulher pm = new PessoaMulher(50.3,  
                                     1.56, 45);  
  
int id1 = p.getIdade(), id2 = pm.getIdade();  
  
int k1 = p.getIMC(), k2 = pm.getIMC();  
  
String s1 = p.classificaObesidade(),  
        s2 = pm.classificaObesidade();
```

# Usando as classes

```
Pessoa p = new Pessoa(72.5, 1.73, 52);  
PessoaMulher pm = new PessoaMulher(50.3,  
                                     1.56, 45);  
int id1 = p.getIdade(), id2 = pm.getIdade();  
int k1 = p.getIMC(), k2 = pm.getIMC();  
String s1 = p.classificaObesidade(),  
        s2 = pm.classificaObesidade();
```

Objeto p é do tipo Pessoa.  
Não possui um método classificaObesidade().

## Exemplo 2

- Elabore uma classe *ContaBancaria*, com os seguintes membros:
  - atributos: nome do cliente, número da conta, saldo;
  - métodos: getters e setters (apenas os necessários), deposita, saca
- Agora acrescente ao projeto duas classes herdadas de *ContaBancaria*: *ContaPoupanca* e *ContaEspecial*, com as seguintes características:

## Exemplo 2

- Classe *ContaPoupança*:
  - atributo dia de rendimento, método *calcularNovoSaldo*, recebe a taxa de rendimento da poupança e atualiza o saldo, novos getters e setters.
- Classe *ContaEspecial*:
  - atributo limite indica quanto a conta pode ficar negativa, redefinição do método sacar, permitindo saldo negativo até o valor do limite, novos getters e setters.

## Exemplo 2

- Após a implementação das classes acima, você deverá implementar uma classe Contas.Java, contendo o método main e que gerencia várias contas. Crie um menu para:
  - a) Incluir uma nova conta
  - b) Sacar um determinado valor de uma conta
  - c) Depositar um determinado valor em uma conta
  - d) Atualizar o valor de todas as contas poupança de um determinado dia
  - e) Mostrar o saldo de cada uma das contas



# Classe Base

- Na classe base, colocamos os membros comuns

```
public class ContaBancaria {  
    private String nomeCliente;  
    private int numConta;  
    private double saldo;
```

# Classe base

```
public ContaBancaria(String n, int num) {  
    nomeCliente = n;  
    numConta = num;  
    saldo = 0.0;  
}  
  
public double getSaldo() {  
    return saldo;  
}  
  
protected void setSaldo(double s) {  
    saldo = s;  
}
```

# Classe base

```
public ContaBancaria(String n, int num) {  
    nomeCliente = n;  
    numConta = num;  
    saldo = 0.0;  
}  
  
public double getSaldo() {  
    return saldo;  
}  
  
protected void setSaldo(double s) {  
    saldo = s;  
}
```

Um membro protected não pode ser acessado por qualquer classe. Mas pode ser acessado por uma subclasse.

# Classe base

```
public void deposita(double qto) {  
    saldo += qto;  
}
```

# Classe base

```
public void deposita(double qto) {  
    saldo += qto;  
}  
  
public void saca(double qto) throws  
IllegalArgumentException {  
    if ( saldo < qto)  
        throw new IllegalArgumentException("Saldo  
insuficiente para esse saque");  
    saldo -= qto;  
}
```

# Classe base

```
public void deposita(double qto) {  
    saldo += qto;  
}
```

```
public void saca(double qto) throws  
IllegalArgumentException {  
    if ( saldo < qto)  
        throw new IllegalArgumentException("Saldo  
insuficiente para esse saque");  
    saldo -= qto;  
}
```

**Note a mensagem adicionada à Exception**

# Subclasse

```
public class ContaPoupanca extends ContaBancaria {  
    private int vencimento;  
}
```

# Subclasse

```
public class ContaPoupanca extends ContaBancaria {  
    private int vencimento;  
}
```

Esse atributo é novo. Só existe nesse tipo de conta.



# Subclasse – construtor

```
public class ContaPoupanca extends ContaBancaria {  
    private int vencimento;
```

```
    public ContaPoupanca(String n, int num, int dia) {  
        super(n, num);  
        vencimento = dia;  
    }
```

# Subclasse – construtor

```
public class ContaPoupanca extends ContaBancaria {  
    private int vencimento;
```

```
    public ContaPoupanca(String n, int num, int dia) {  
        super(n, num);  
        vencimento = dia;  
    }
```

Antes de tudo, chama o construtor da superclasse.

# Subclasse – construtor

```
public class ContaPoupanca extends ContaBancaria {  
    private int vencimento;
```

```
    public ContaPoupanca(String n, int num, int dia) {  
        super(n, num);  
        vencimento = dia;  
    }
```

Antes de tudo, chama o construtor da superclasse.

Se não houver uma chamada explícita, é feita chamada a `super()`, ou seja construtor sem parâmetros.

# Subclasse – métodos novos

```
public int getVencimento() {  
    return vencimento;  
}
```

```
public void setVencimento(int vencimento) {  
    this.vencimento = vencimento;  
}
```

Refere-se ao objeto que foi usado para fazer a chamada do método.

# Subclasse – métodos novos

```
public void atualiza(double taxa) {  
    double s = getSaldo();  
    setSaldo(s * (1.0 + taxa));  
}
```

Não queremos os atributos da superclasse sendo acessados diretamente na subclasse

# Subclasse 2

```
public class ContaEspecial extends ContaBancaria {  
    private double limite;  
  
    public ContaEspecial(String n, int num, double l) {  
        super(n, num);  
        limite = l;  
    }  
}
```

# Subclasse 2 – método “novo”

```
@Override
```

```
public void saca(double qto) {
    double s = getSaldo();
    if ( qto > s + limite )
        throw new IllegalArgumentException("Limite
excedido para esse saque");
    setSaldo( s - qto );
}
```

# Subclasse 2 – método “novo”

@Override

```
public void saca(double qto) {
    double s = getSaldo();
    if ( qto > s + limite )
        throw new IllegalArgumentException("Limite
excedido para esse saque");
    setSaldo( s - qto );
}
```

Esse método substitui o método ContaBancaria.saca.  
Por isso usamos @Override



# @Override

- Já usamos antes. Quando?

# @Override

- Já usamos antes. Quando?
- Ao definirmos o método `toString` para nossas classes.
- Em Java toda classe herda da classe **Object**.
- Nessa classe são definidos alguns métodos inclusive o `toString()`.
- <http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

# Hierarquia

