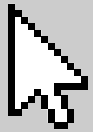
PUC Minas

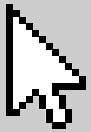
Programação Modular



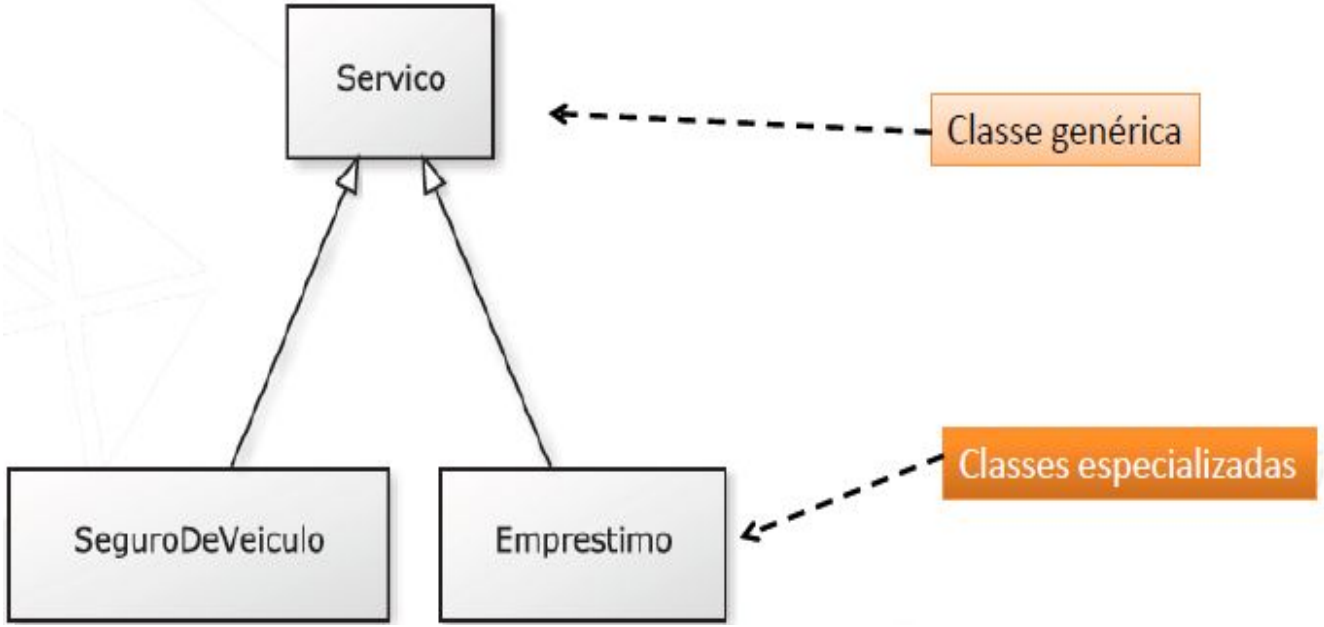
PUC Minas
Bacharelado em Engenharia de Software
Prof. Daniel Kansaon
slides adaptados do Prof. Danilo Boechat

PUC Minas

HERANÇA



Herança


Servico
Classe genérica
SeguroDeVeiculo
Emprestimo
Classes especializadas

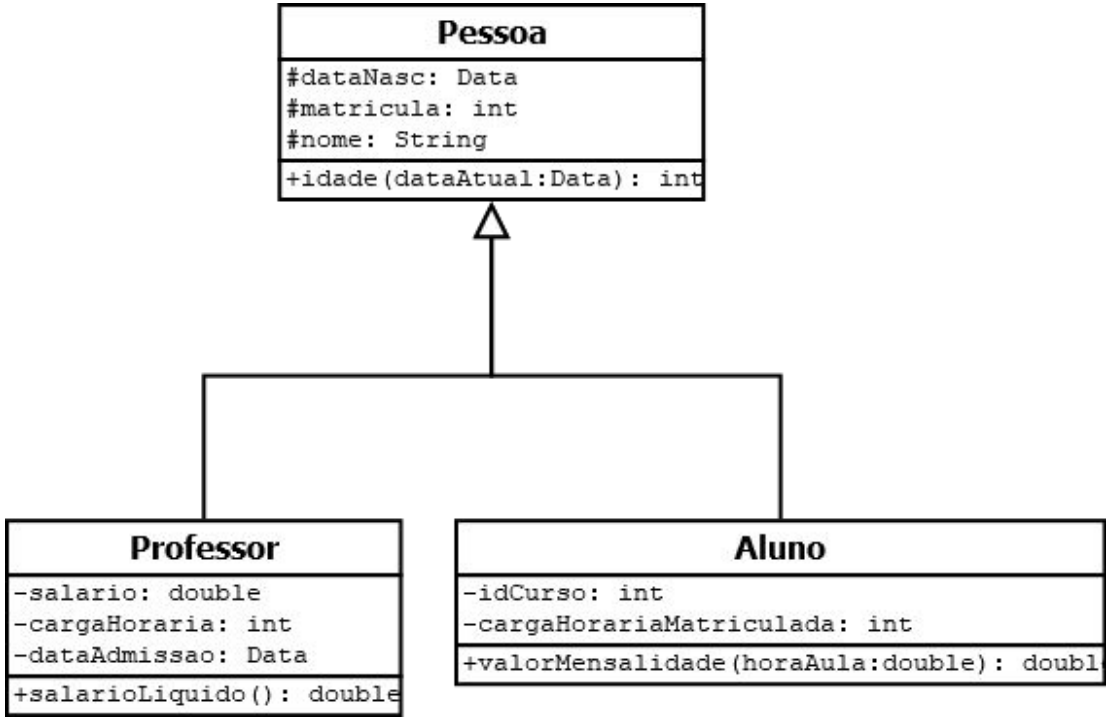
Herança

Herança permite que uma classe reutilize **atributos** e **métodos** de outra.

A classe que herda é chamada de **subclasse**.

A classe da qual ela herda é chamada de **superclasse**.

Herança


Pessoa
#dataNasc: Data
#matricula: int
#nome: String
+idade(dataAtual:Data): int
Professor
-salario: double
-cargaHoraria: int
-dataAdmissao: Data
+salarioLiquido(): double
Aluno
-idCurso: int
-cargaHorariaMatriculada: int
+valorMensalidade(horaAula:double): doubl

Herança

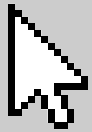
Reutilização de código

Organização: estrutura lógica entre classes.

Polimorfismo

PUC Minas

POLIMORFISMO



Polimorfismo

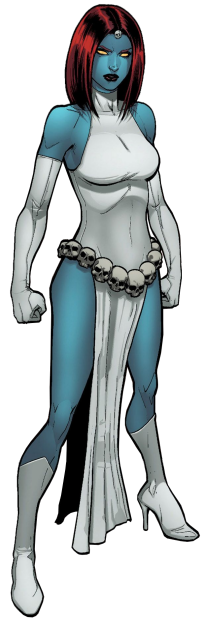
Diferentes comportamentos de métodos com mesma assinatura

Polimorfismo

O termo polimorfismo é originário do grego e significa "***muitas formas***"

- **poli** = muitas
- **morphos** = formas

Tal como a Mística 😅



Polimorfismo

Em outras palavras, o polimorfismo quer dizer que:

- **Que se apresenta ou ocorre sob formas diversas**
- **Que assume ou passa por várias formas, fases etc.**

Polimorfismo

Na POO pode-se dizer também que o objeto pode ter “vários comportamentos”

Quando um método é chamado ele pode se referir a qualquer objeto

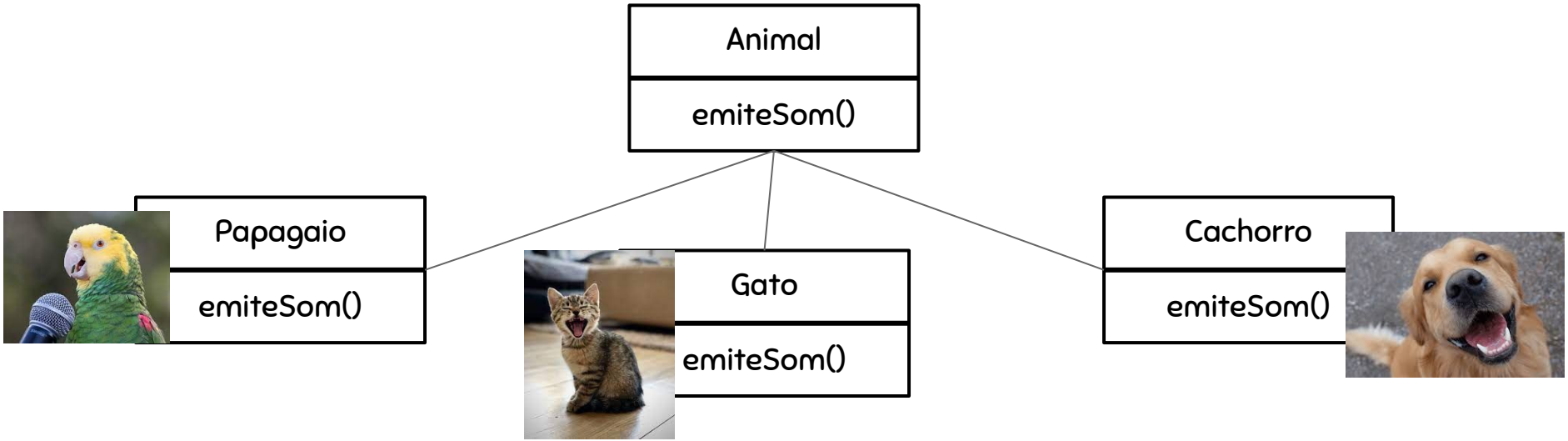
- **Exemplo**: método “***abrir()***”:

ele pode abrir uma tampa de garrafa, uma janela, uma porta, pode abrir alguma coisa

Polimorfismo - Vantagens

- **Instanciar objetos diferentes** para um mesmo tipo
- **Substituir condicionais**:
  - Não é necessário colocar instruções condicionais para definir qual método executar
- Usar parâmetros que aceitam **diferentes tipos** de objetos desde que tenham a mesma herança de classes

Polimorfismo - Exemplo 1



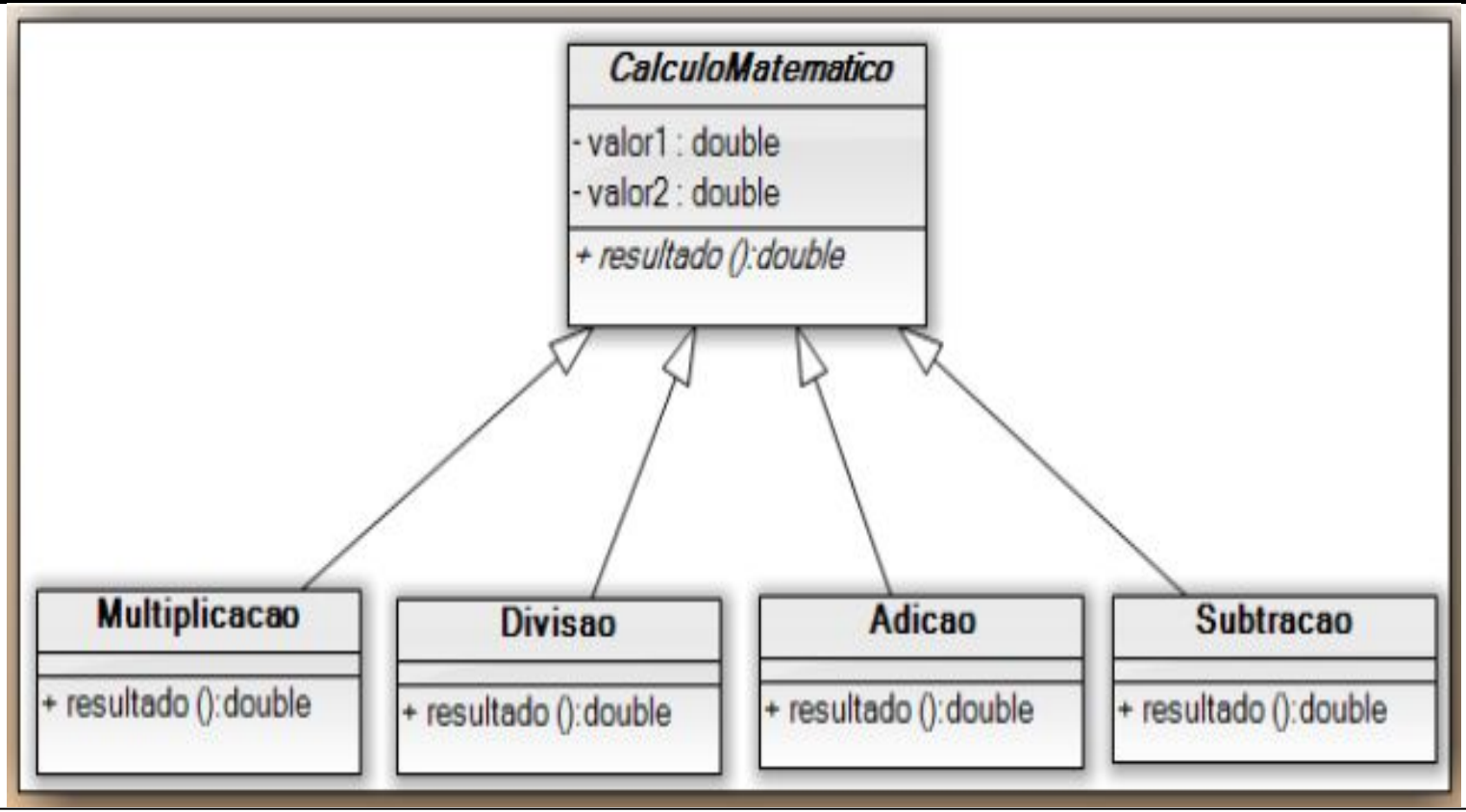
Polimorfismo - Exemplo

```java
class Animal{
    public String emiteSom(){
        return "";
    }
}

class Gato extends Animal{
    @Override
    public String emiteSom(){
      return "Miau";
    }
}

class Cachorro extends Animal{
    @Override
    public String emiteSom(){
        return "Au Au";
    }
}
```

```java
public static void main(String[] args) {
    Animal animal = new Gato();
    System.out.println(animal.emiteSom());
}
```

O que será impresso?

Polimorfismo - Exemplo 2



Polimorfismo - Exemplo 2

Classe abstrata!?

```
public abstract class CalculoMatematico {
    private double Valor1;
    private double Valor2;

    public abstract double resultado();

    public double getValor1(){
        return this.Valor1;
    }

    public double getValor2(){
        return this.Valor2;
    }

    public void setValor1(double valor){
        this.Valor1 = valor;
    }

    public void setValor2(double valor){
        this.Valor2 = valor;
    }
}
```

Polimorfismo - Exemplo 2

```
public class Soma extends CalculoMatematico {
    @Override
    public double resultado(){
      return getValor1()+getValor2();
    }
}
```

Polimorfismo - Exemplo 2

```java
public class Soma extends CalculoMatematico {
    @Override
    public double resultado(){
      return getValor1()+getValor2();
    }
}
```

```java
public class Subtracao extends CalculoMatematico{
    @Override
    public double resultado(){
      return getValor1() - getValor2();
    }
}
```

Polimorfismo - Exemplo 2

```
public class Multiplicacao extends CalculoMatematico{
    @Override
    public double resultado(){
      return getValor1() * getValor2();
    }
}
```

```
public class Divisao extends CalculoMatematico {
    @Override
    public double resultado(){
      return getValor1() / getValor2();
    }
}
```

```java
public class PolimorfismoCalculadora {
    public static void main(String[] args) {
        /* Partindo de um mesmo tipo de objeto, podemos instanciar
         *      os outros tipos.
         * Mas para isso todas as classes devem implementar a
         *      mesma classe pai.
         * Veja que definimos o tipo como CalculoMat
         */
        CalculoMatematico calc;
        //aqui é criado como Adicao
        calc = new Adicao();
        calc.setValor1(23);
        calc.setValor2(34);
        System.out.println(calc.resultado());
        //aqui é criado como Divisao
        calc = new Divisao();
        calc.setValor1(23);
        calc.setValor2(34);
        System.out.println(calc.resultado());
    }
}
```

Partindo de um mesmo tipo de objeto, podemos instanciar os outros tipos. Mas para isso todas as classes devem implementar a mesma classe Pai.

```
Saída - PolimorfismoCalculadora (run)
run:
57.0
0.6764705882352942
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

Polimorfismo

Utilizado quando métodos herdados de uma mesma classe possuem comportamentos diferentes

O polimorfismo é alcançado com auxílio do uso de **herança nas classes** e a **reescrita** (***overriding***) de métodos das superclasses nas suas subclasses

Outra forma de se utilizar o polimorfismo é **sobrecarregando** (***overloading***) os métodos

Sobrescrita e Sobrecarga

Sobrescrita

@Override

A sobrescrita de métodos permite criar um novo método na classe filha, contendo **a mesma assinatura e mesmo tipo de retorno** do método sobrescrito

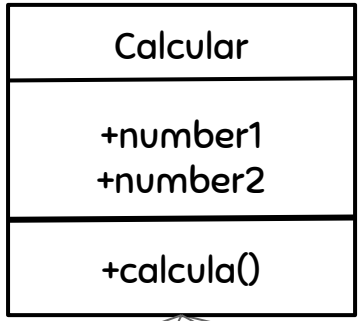
Override - Exemplo 1

```
class Animal{
    public String emiteSom(){
        return "";
    }
}

class Gato extends Animal{
    @Override
    public String emiteSom(){
      return "Miau";
    }
}

class Cachorro extends Animal{
    @Override
    public String emiteSom(){
        return "Au Au";
    }
}
```

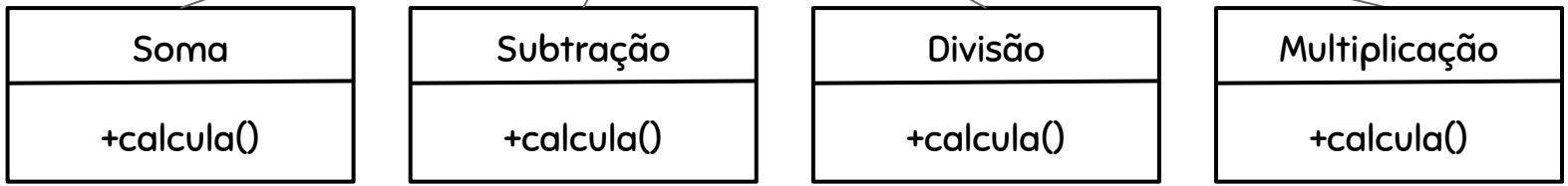
```
public static void main(String[] args) {
    Animal animal = new Gato();
    System.out.println(animal.emiteSom());
}
```

Override - Exemplo 2

O método +calcula() é diferente em cada uma das subclasses




```java
public class Soma extends Calcular{

    public double calcula(){

        return (this.number1 + this.number2)

    }
}
```



Cada classe filha se sobrepõe (***overrides***) à da classe mãe

Sobrecarga

@Overload

A sobrecarga permite criar vários métodos com mesmo nome, mas com **assinaturas diferentes**, ou seja, com argumentos diferentes

Sobrecarga - Exemplo 1

```
public class Calculadora {

    public int soma(int num1, int num2) {

        return num1 + num2;

    }

    public double soma(double num1, double num2) {

        return num1 + num2;

    }
}
```

⇒ Assim, evitamos poluir o código com nomes de métodos desnecessários

⇒ Variamos os parâmetros que recebemos e o retorno

⇒ Utilizamos o mesmo nome (assinatura) do método

Sobrecarga - Exemplo 1

```
public class Calculadora {

    public int soma(int num1, int num2) {

        return num1 + num2;

    }

    public double soma(double num1, double num2) {

        return num1 + num2;

    }
}
```

Exemplos ruins:

- `double somaDouble(double num1, double num2)`
- `int somaInt(int num1, int num2)`

Sobrecarga

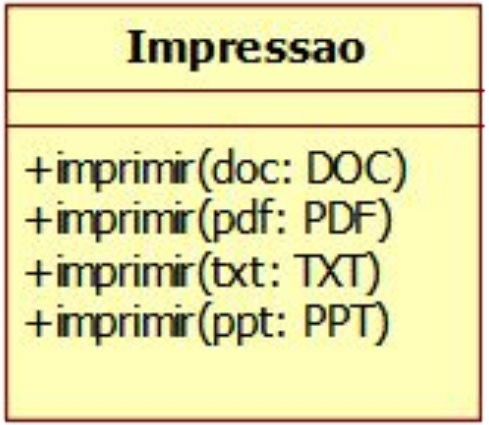
O polimorfismo nos permite decidir qual dos métodos será usado em tempo de execução

Podemos decidir qual dos métodos será usado, em tempo de compilação, distinguindo pelos parâmetros passados

Sobrecarga - Exemplo 2

```
public class Produto {

    public void buscar(int codigo) {

        System.out.println("Produto: " + codigo);

    }

    public void buscar(String nome) {

        System.out.println("Produto: " + nome);

    }
```

Sobrecarga - Exemplo 3



É possível substituir condicionais com o uso de polimorfismo!

```
public class Calculadora {
    /** Neste método temos que informar o tipo
     de calculo que o mesmo irá fazer
     */
    public static double caloular(TipoCalculo tipo, double valor1, double valor2)
        double resultado = 0;
        //e aqui temos o condicional switch
        switch (tipo) {
            case Adicao:
                resultado = valor1 + valor2;
                break;
            case Subtracao:
                resultado = valor1 - valor2;
                break;
            case Multiplicacao:
                resultado = valor1 * valor2;
                break;
            case Divisao:
                resultado = valor1 / valor2;
                break;
            default:
                throw new Exception("nunca poderá chegar aqui");
        }
        return resultado;
    }
```

O método **calcular** necessita de um parâmetro que é o tipo de operação.
A partir do tipo é usado um condicional *switch* para avaliar a operação.
Já o método sobrecarregado **calcular** do próximo slide, não requer esta avaliação, pois usa o polimorfismo para executar a operação correta.

Acessando os dois métodos **calcular** para demonstrar a chamada onde é necessário informar o tipo de operação, e onde é necessário simplesmente a **Instância** do tipo para definir o cálculo.

```
11  public class PolimortismoCalculadora {
12
13      public static void main(String[] args) {
14          //aqui vamos usar o condicional
15          System.out.println("Usando o Condicional");
16          System.out.println(Calculadora.calcular(TipoCalculo.Adicao, 23, 34));
17          System.out.println(Calculadora.calcular(TipoCalculo.Divisao, 23, 34));
18          System.out.println(Calculadora.calcular(TipoCalculo.Multiplicacao, 23, 34));
19          System.out.println(Calculadora.calcular(TipoCalculo.Subtracao, 23, 34));
```

REFAZENDO...

```
...

    /// fazendo deste modo não precisamos
    /// utilizar nenhum condicional
    public static double calcular(CalculoMatematico calc) {
        return calc.resultado();
    }
}
```

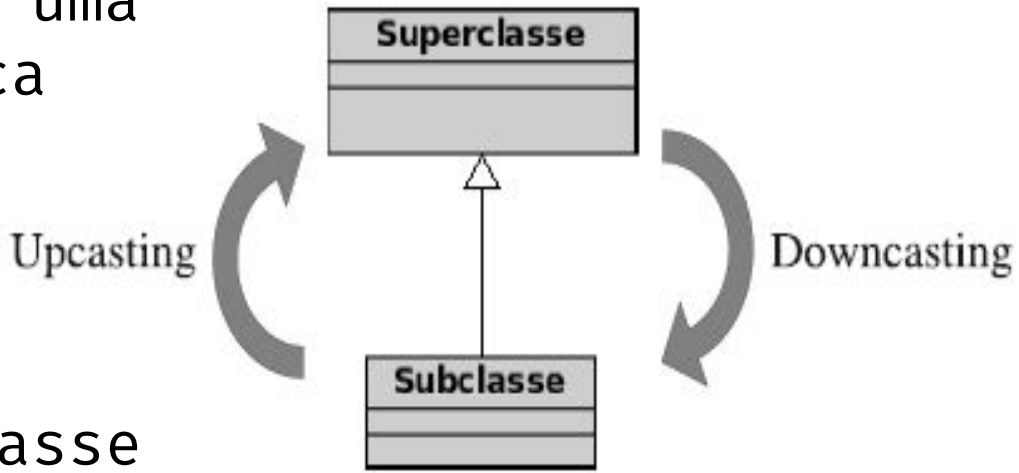
Este método **calcular** recebe como parâmetro uma instância de **calculoMatematico**. Logo, será chamado o método **resultado()** da instância apropriada, podendo ser uma Divisão, Multiplicacao, Subtracao ou Adicao – tudo depende da instância que será passada no parâmetro.

```
System.out.println("Usando polimorfismo");
//Aqui nao iremos usar o condicional.
//substituiremos pelo uso do polimorfismo
CalculoMatematico calc;
//aqui é criado como Adicao
calc = new Adicao();
calc.setValor1(23);
calc.setValor2(34);
System.out.println(Calculadora.calcular(calc));
//aqui é criado como Divisao
calc = new Divisao();
calc.setValor1(23);
calc.setValor2(34);
System.out.println(Calculadora.calcular(calc));
//aqui é criado como Multiplicacao
calc = new Multiplicacao();
calc.setValor1(23);
calc.setValor2(34);
System.out.println(Calculadora.calcular(calc));
//aqui é criado como Subtracao
calc = new Subtracao();
calc.setValor1(23);
calc.setValor2(34);
System.out.println(Calculadora.calcular(calc));
```

Polimorfismo - Conversões

Downcast

⇒ converter em uma classe mais específica



Upcast

⇒ definir em uma classe mais genérica

Upcast - Exemplo

```
class Animal{
    public String emiteSom(){
        return "";
    }
}

class Gato extends Animal{
    @Override
    public String emiteSom(){
      return "Miau";
    }
}

class Cachorro extends Animal{
    @Override
    public String emiteSom(){
        return "Au Au";
    }
}
```

```
public static void main(String[] args) {
    Gato gato = new Gato();
    Animal animal = gato;
}
```

```java
public class PolimorfismoCalculadora {

    public static void main(String[] args) {
        //definir objeto
        CalculoMatematico calc = new Adicao();
        calc.setValor1(22);
        calc.setValor2(33);

        //downcast (classe mais especifica)
        Adicao adicao = (Adicao) calc;
        System.out.println(adicao.resultado());

        //upcast (classe mais genérica)
        calc = (CalculoMatematico)adicao;
        //calc = adicao;
        System.out.println(calc.resultado());
    }
}
```

Downcast: atribuição de uma instância de uma classe mais genéric para uma mais específica.

Upcast: atribuição de uma instância de uma classe mais Específica para uma mais genéria.

```java
public class PolimorfismoCalculadora {

    public static void main(String[] args) {
        //definir objeto
        CalculoMatematico calc = new Adicao();
        calc.setValor1(22);
        calc.setValor2(33);

        //downcast (classe mais especifica)
        Adicao adicao = (Adicao) calc;
        System.out.println(adicao.resultado());

        //upcast (classe mais genérica)
        //calc = (CalculoMatematico)adicao;
        calc = adicao;
        System.out.println(calc.resultado());
    }
}
```

A operação de *upcast* pode ser realizada desta maneira.

Downcast- Exemplo

```
class Animal{
    public String emiteSom(){
        return "";
    }
}

class Gato extends Animal{
    @Override
    public String emiteSom(){
      return "Miau";
    }
}

class Cachorro extends Animal{
    @Override
    public String emiteSom(){
        return "Au Au";
    }
}
```

```
public static void main(String[] args) {
    Gato gato = new Gato();
    Animal animal = gato;

    gato = (Gato) animal;
}
```

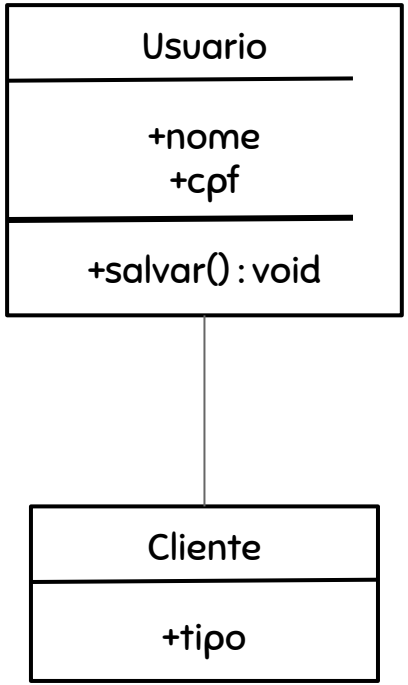
Downcast- Exemplo

```java
class Animal{
    public String emiteSom(){
        return "";
    }
}

class Gato extends Animal{
    @Override
    public String emiteSom(){
      return "Miau";
    }
}

class Cachorro extends Animal{
    @Override
    public String emiteSom(){
        return "Au Au";
    }
}
```

```java
public static void main(String[] args) {
    Animal animal = new Animal();

    Gato g  = (Gato) animal;
}
```

⚠️ Exemplo de downcast perigoso

Exemplo

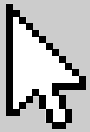


Exemplo

```
class SistemaCadastro {
        public static void cadastrarUsuario(Usuario u) {
         u.salvar();
    }
}

public static void main(String[] args) {
    Cliente cliente = new Cliente("Joao", "323.234.135.-06", "vip");
    SistemaCadastro.cadastrarUsuario(cliente);
}
```

PUC Minas

Obrigado!



PUC Minas
Bacharelado em Engenharia de Software
Prof. Danilo Boechat Seufitelli
daniloboechat@pucminas.br