# Programação para Web I
# Modificadores

Profa. Silvia Bertagnolli

# Modificadores de Acesso

+ public

- private

# protected

~ default

final

static

abstract

native

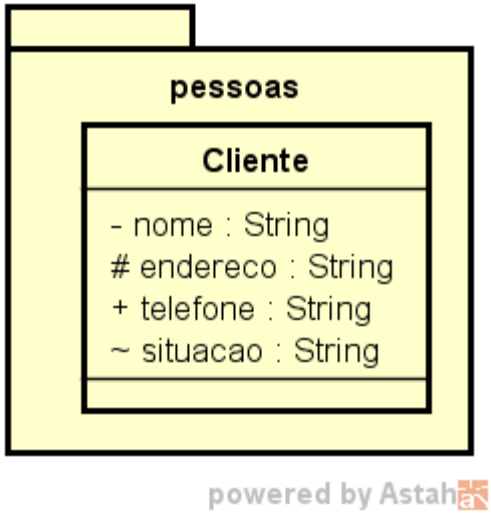
synchronized

volatile

# VISIBILIDADE

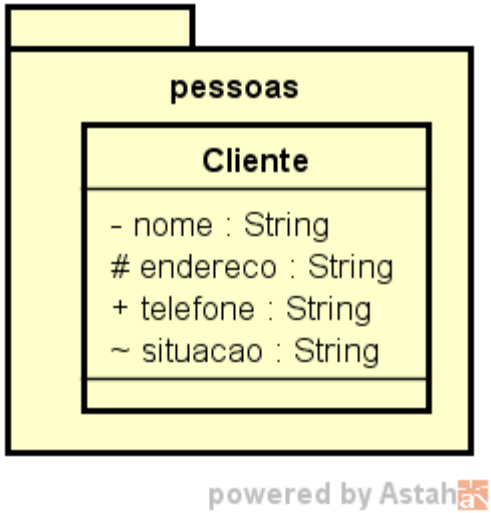
| modificador | private | default | protected | public |
|---|---|---|---|---|
| Na classe | sim | sim | sim | sim |
| Outras classes | não | sim | sim | sim |
| Classes no mesmo pacote | não | sim | sim | sim |
| Subclasses em pacotes diferentes | não | não | sim | sim |

# Exemplo: visibilidade



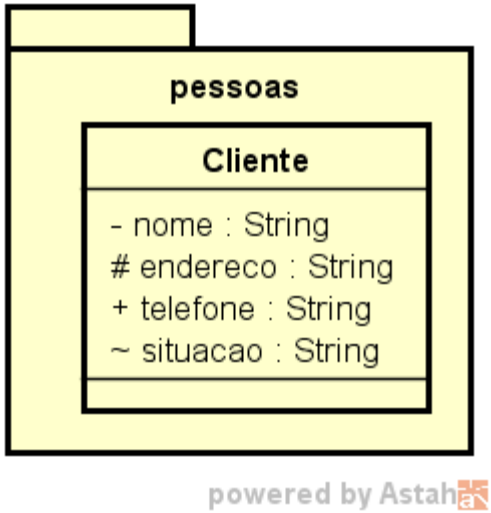

```
package pessoas;

public class Cliente{

        private String nome;

         protected String endereco;

        public String telefone;

        String situacao;

...

}
```

# Exemplo: visibilidade




```
package pessoas;
public class Teste{
  public static void main(String args []){
        Cliente cliente = new Cliente();
         System.out.println(cliente.nome);
         System.out.println(cliente.endereco);
        System.out.println(cliente.telefone);
         System.out.println(cliente.situacao);
  }
}
```

# Exemplo: visibilidade




```
package teste;
public class Teste{
  public static void main(String args []){
        Cliente cliente = new Cliente();
         System.out.println(cliente.nome);
         System.out.println(cliente.endereco);
        System.out.println(cliente.telefone);
         System.out.println(cliente.situacao);
  }
}
```

# Resumo

| Modificador/Elemento | Classe | Método | Atributo |
|---|---|---|---|
| public | ✓ | ✓ | ✓ |
| private | ✗ | ✓ | ✓ |
| protected | ✗ | ✓ | ✓ |

# Modificador final

# Final pode ser aplicado com:

- Classe - classe final
- Método - método final
- Atributos - constantes
- Variáveis que sempre vão apontar para o mesmo objeto

# Classe final

Classe atingiu o nível máximo de especialização e não poderá mais ser especializada - nenhuma outra classe jamais poderá estender esta classe

Quando usar? Garantir que nenhum método da classe será sobreposto

Exemplo: java.lang.String

# Classe final

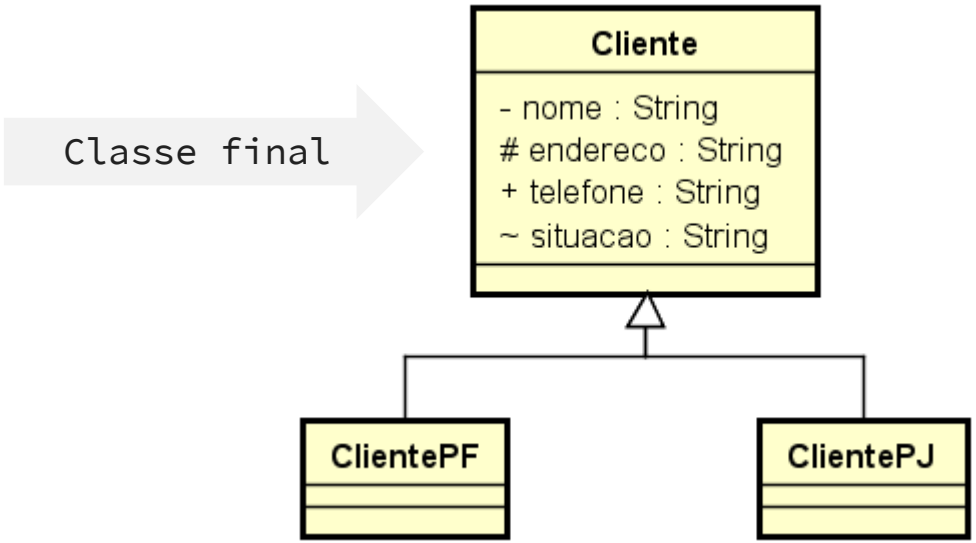
Vantagens:

- Permite **proteger** um código
- Aumenta o desempenho do código

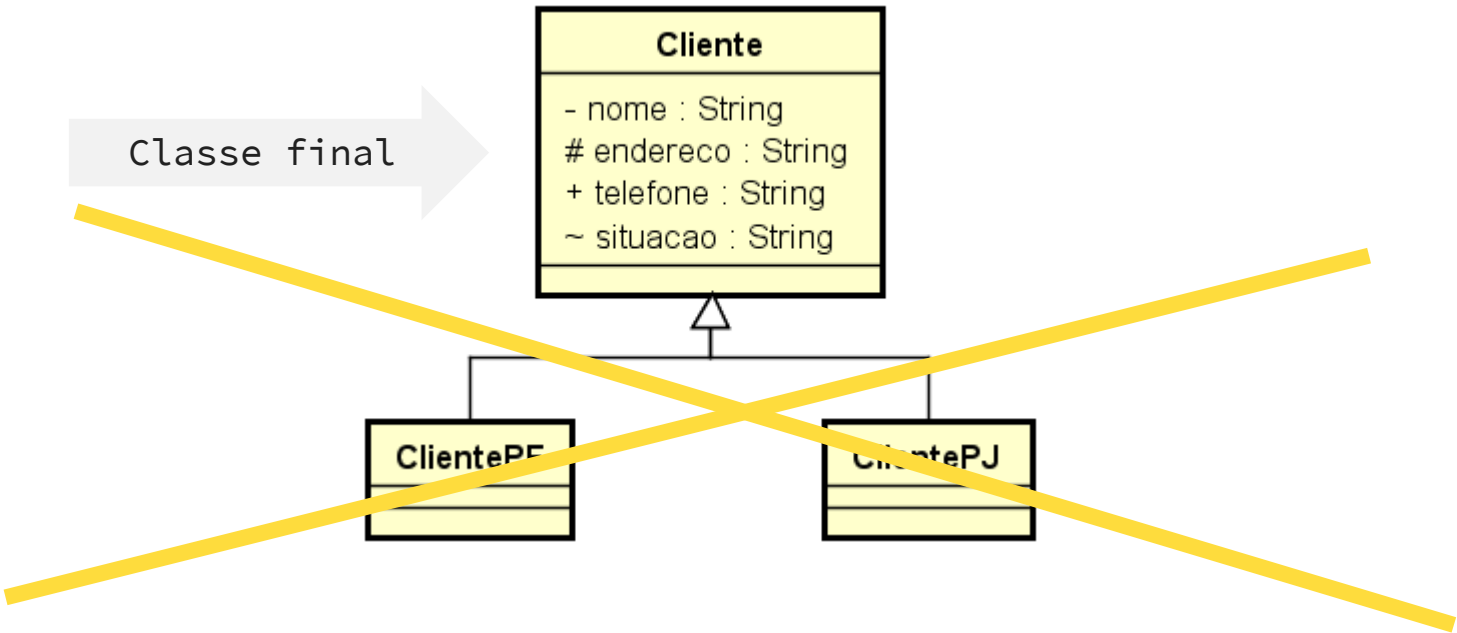
Desvantagem - **reduz** as possibilidades de **herança**

# Classe final

Nenhuma outra classe jamais poderá estender esta classe



# Classe final



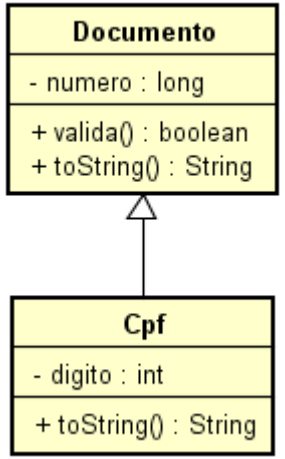
# Classe final: sintaxe

```
<modificador> final class <nome_classe>{

}
```

# Classe final

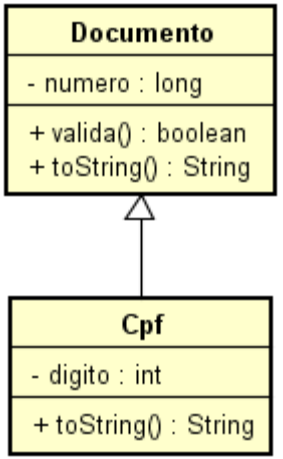
```
public final class Documento{
     public boolean valida() {
               // corpo método
               return true;
      }
}

public class Cpf extends Documento{
....
}
```

?



# Classe final

```
public class TesteFinal{

        public static void main(String args[]) {

            Documento d = new Documento();

            if(d.valida())

               System.out.println("Documento válido");

       }

}
```



# Método final

Método que **não** pode ser **sobrescrito** nas subclasses

Isso oferece **segurança** e **proteção**

Método declarado como final terá o seu protótipo sempre como foi definido e quando chamado por outros objetos seu código será executado

**O que é sobrescrita e o que é sobrecarga?**

# Método final

O desempenho de execução de um método final é maior, pois as chamadas são substituídas pelo código contido na definição do método

“[...] se um método possuir uma especificação bem definida e não for sofrer especializações/redefinições pelas classes herdeiras, é aconselhável que o mesmo receba o modificador final por razões de segurança e desempenho.”

# Método final: sintaxe

```
<modificador> final <tipo_retorno>
            <nome_metodo> (<lista_parâmetros>){
             //...
    }
```

# Método final

```
public class Cpf{

        private long numero;
        private int digito;
        public final boolean valida() {
                // corpo método
                return true;
        }
        // métodos get/set

}
```

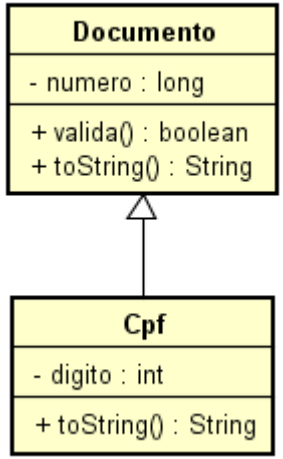
# Método final

```
public class TesteFinal2{

       public static void main(String args[]) {

          Cpf c = new Cpf();

          if(c.valida())

              System.out.println("Cpf é válido");

    }

}
```

# Método final

```
public class Documento{
    //atributos
        public final boolean valida() { ... }
}
public class Cpf extends Documento{
      //atributos
      public boolean valida() {
            // corpo método
     }
}
```

?



# Atributo final

Conhecido como **constante** dos objetos de uma classe

**Cuidado**! Ao declarar uma variável final é necessário fornecer um valor explícito

Em Java nomenclatura: **todas** letras em **maiúsculas**

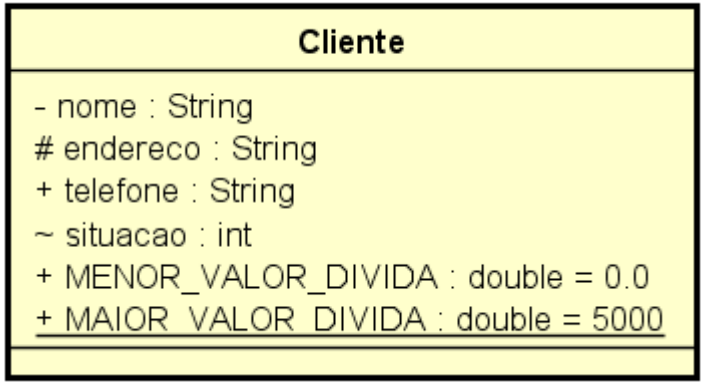
# Atributo final: sintaxe

```
<modificador> final <tipo> <nome_variável> = valor;
```

ou:

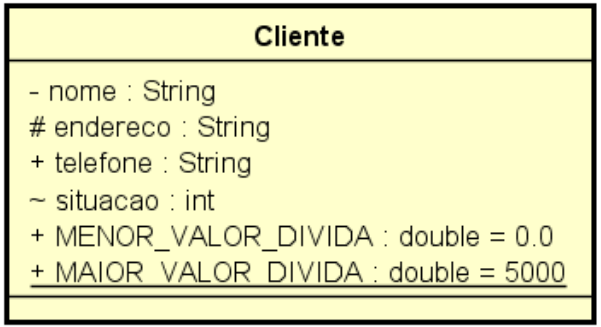
```
<modificador> static final <tipo> <nome_variável> = valor;
```

# Atributo final




```
public class Cliente{

    //…

    public final double MENOR_VALOR_DIVIDA = 0.0;

    public static final double MAIOR_VALOR_DIVIDA = 5000.0;


   //…

}
```

# Atributo final




```
public class TesteFinal3{

        public static void main(String args[]) {

         Cliente c = new Cliente();

         System.out.println(c.MENOR_VALOR_DIVIDA);

         System.out.println(Cliente.MAIOR_VALOR_DIVIDA);

    }

}
```

# Variáveis final

Neste caso as variáveis sempre vão apontar para o mesmo objeto

**Exemplo 1**

```
public void metodo1(final String valor){
    valor = “outro objeto String”;
}
```

Erro do compilador: A variável não pode ser reatribuída a outro objeto

**Exemplo 2**

```
public void metodo2(){
    final String novoValor;
    novoValor = “outro objeto String”;
}
```

# Resumo

| Modificador/Elemento | Classe | Método | Atributo |
|---|---|---|---|
| public | ✓ | ✓ | ✓ |
| private | x | ✓ | ✓ |
| protected | x | ✓ | ✓ |
| final | ✓ | ✓ | ✓ |

# Modificador static

# STATIC PODE SER APLICADO COM:

- Método – método de classe
- Atributos – variável de classe

- Importações estáticas: import static

# STATIC

Recursos estáticos **pertencem** a uma **classe** e **não** estão associados a uma **instância**

Denominados:

- Atributos estáticos ou variáveis de classe
- Método estáticos ou métodos de classe
- Classe estática - quando é classe interna

# Variável de Classe

Apenas uma cópia (classe) para todos as instâncias da classe

Exemplos:

java.lang.Math.E (2.71828...)

java.lang.Math.PI (3.14159...)

**Existe alguma constante definida na classe Integer?**

**Module** java.base

**Package** java.lang

## Class Integer

java.lang.Object
    java.lang.Number
        java.lang.Integer

**All Implemented Interfaces:**

`Serializable`, `Comparable<Integer>`, `Constable`, `ConstantDesc`

---

```
public final class Integer
extends Number
implements Comparable<Integer>, Constable, ConstantDesc
```

### *Field Summary*

**Fields**

| Modifier and Type | Field | Description |
|---|---|---|
| `static int` | `BYTES` | The number of bytes used to represent an `int` value in two's complement binary form. |
| `static int` | `MAX_VALUE` | A constant holding the maximum value an `int` can have, $2^{31}-1$. |
| `static int` | `MIN_VALUE` | A constant holding the minimum value an `int` can have, $-2^{31}$. |
| `static int` | `SIZE` | The number of bits used to represent an `int` value in two's complement binary form. |
| `static Class<Integer>` | `TYPE` | The `Class` instance representing the primitive type `int`. |

# Variável de Classe: sintaxe

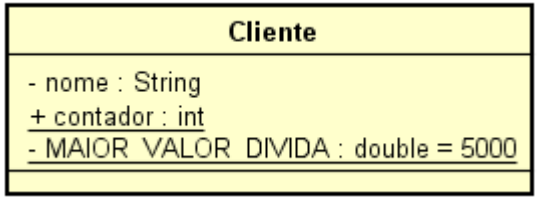
```
<modificador> static <tipo> <nome_variável>;
```

ou:

```
<modificador> static final <tipo> <nome_variável>;
```

# Variável de Classe

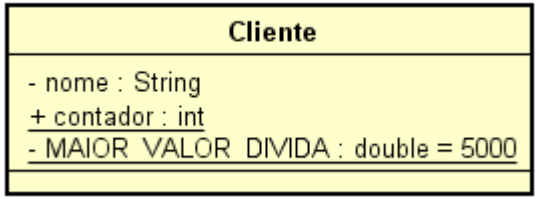
```
public class Cliente{
    //…
    public static int contador = 0;
    public static final double MAIOR_VALOR_DIVIDA = 5000.0;
   //…
}
```




**Vamos adicionar o contador na classe Cliente**

# Variável de Classe

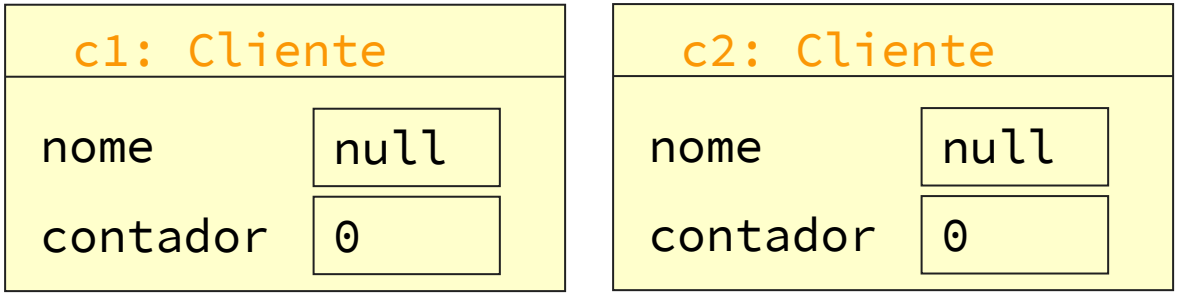
```
public class Cliente{
    //…
    public static int contador = 0;
    public Cliente(){ this(null);}
    public Cliente(String nome) {
          contador++;
          this.nome = nome;
     }
     //outras definições
}
```




| Tipo | Valor |
|---|---|
| Tipo inteiro | 0 |
| TipoPF (float e double | 0.0 |
| boolean | False |
| QQ classe | null |

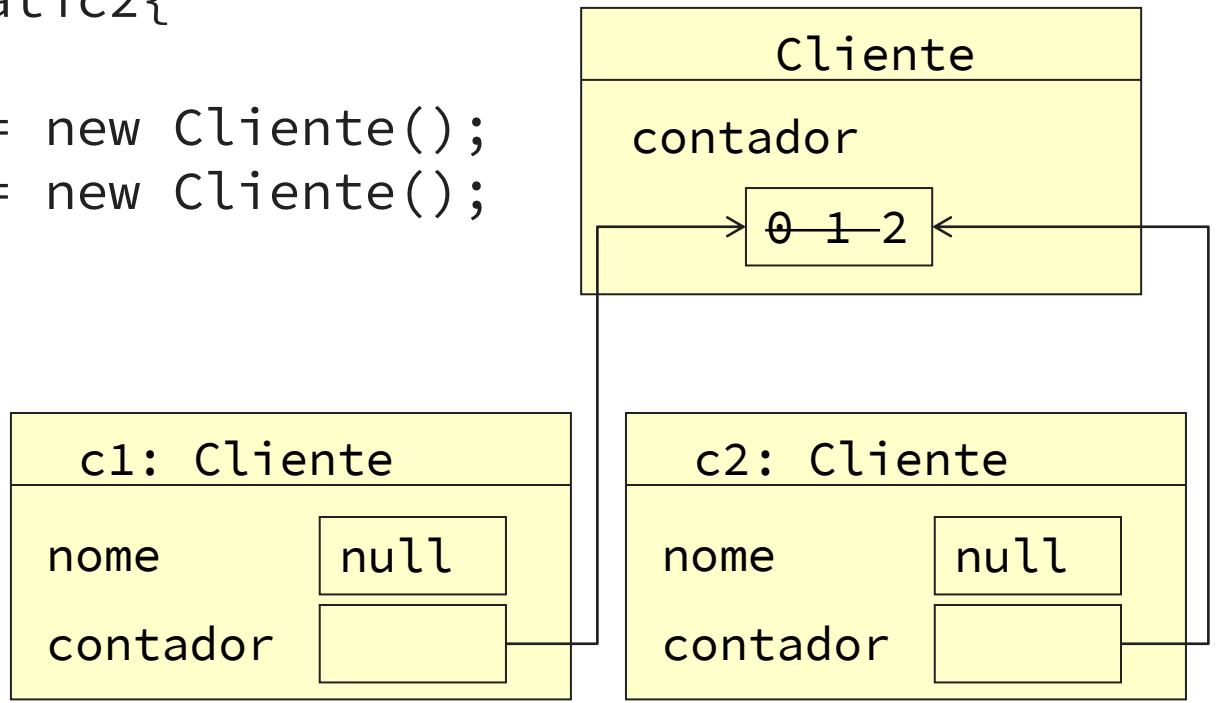
# Se contador não fosse Variável de Classe

```
public class TesteStatic1{
  ... main(...){
          Cliente c1 = new Cliente();
          Cliente c2 = new Cliente();
      }
}
```



# Se contador fosse Variável de Classe

```
public class TesteStatic2{
  ... main(...){
          Cliente c1 = new Cliente();
          Cliente c2 = new Cliente();
      }
}
```



# Variável de Classe

```
public class TesteStatic2{
 ... main(...){
          Cliente c1 = new Cliente();
          Cliente c2 = new Cliente();
          c1.contador;
          //ou:
          Cliente.contador;
     }
}
```

Pode ser acessado usando um objeto qualquer ou usando o nome da classe

# Método de Classe

Não tem permissão para usar os recursos não estáticos definidos em sua classe:

- Acessar/usar diretamente variáveis de instância
- Chamar diretamente métodos de instância

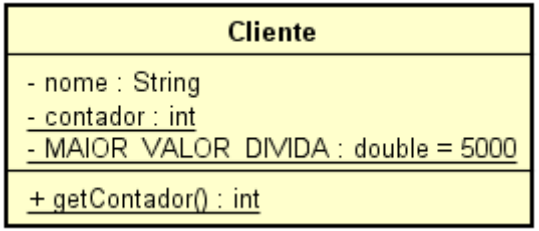
Exemplo: static void main

# Método de Classe: sintaxe

```
<modificador> static <tipo_retorno>
                              <nome> (<lista_parâmetros>){
     //...
 }
```

# Método de Classe

```
public class Cliente{

    //…

    private static int contador = 0;

    //…

    public static int getContador(){

        return contador;

   }

}
```




Vamos declarar o contador como private e definir o método getContador() na classe Cliente

# Método de Classe

```
public class TesteStatic3{
  ... main(...){
          Cliente c1 = new Cliente();
          Cliente c2 = new Cliente();
          int cont1= c1.getContador();
          //ou:
          int cont2 = Cliente.getContador();
      }
}
```

Pode ser acessado usando um objeto qualquer ou usando o nome da classe

# Importações estáticas

A partir do J2SDK 5.0 o comando import foi aprimorado para permitir a importação de métodos e variáveis de classe

Exemplo:

```
import static java.lang.System.*;
```

Isso permitirá usar métodos e campos estáticos da classe System sem a necessidade de usar como prefixo o nome da classe:

- System.out.println();
- out.println();

# Resumo

| Modificador/Elemento | Classe | Método | Atributo |
|---|---|---|---|
| public | ✓ | ✓ | ✓ |
| private | x | ✓ | ✓ |
| protected | x | ✓ | ✓ |
| final | ✓ | ✓ | ✓ |
| static | ✓* | ✓ | ✓ |
| static final | x | x | ✓ |

*** Usado somente para classes internas**

# Modificador abstract

# ABSTRACT PODE SER APLICADO COM:

- Classe – classe abstrata
- Métodos – método abstrato

# Classe abstrata

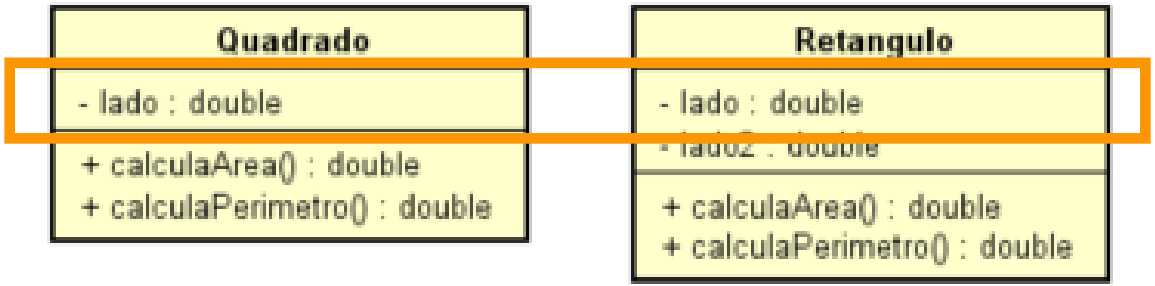
A única finalidade é ser estendida

Incompleta - geralmente, contém métodos abstratos

Métodos **podem** ser definidos nas subclasses

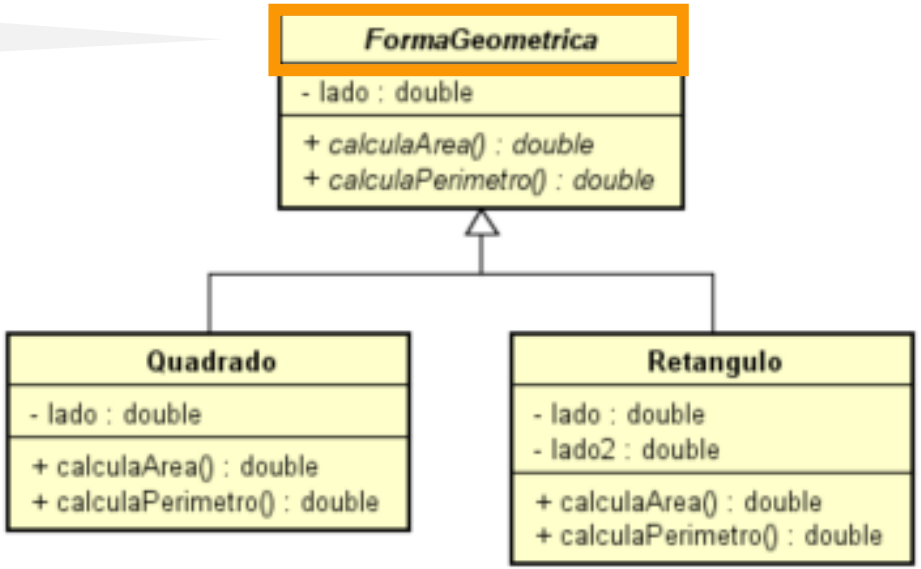
**Obs.:** se um método for definido como abstrato dentro de uma classe toda a classe deverá ser declarada como abstrata

# Classe abstrata



# Classe abstrata

Nome em itálico ou com o esteriótipo <<abstract>>



# Classe abstrata: sintaxe

```
<modificador> abstract class <nome_classe>{

        //...

}
```

# Classe abstrata

```
public abstract class FormaGeometrica{

    //…

    public FormaGeometrica(){}

}
```

# Classe abstrata

```
public class Teste7{
    public static void main(…){
        FormaGeometrica forma1 = new FormaGeometrica();
        FormaGeometrica forma2 = new Quadrado();
        FormaGeometrica vet[] = new FormaGeometrica[10];
        vet[0] = new FormaGeometrica();
        vet[1] = new Quadrado();
        vet[2] = new Retangulo();
        forma1.calculaArea();
    }
```

# Classe abstrata

```
public class Teste7{
    public static void main(…){
        FormaGeometrica forma1 = new FormaGeometrica();x
        FormaGeometrica forma2 = new Quadrado();
        FormaGeometrica vet[] = new FormaGeometrica[10];
        vet[0] = new FormaGeometrica();x
        vet[1] = new Quadrado();
        vet[2] = new Retangulo();
        forma1.calculaArea();
    }
```

# Método abstrato

Método **declarado**, mas **não** foi **implementado**

**Incompleto**: falta o corpo

Um método é declarado abstrato quando for significativo para a classe derivada e a implementação não é significativa para a classe base

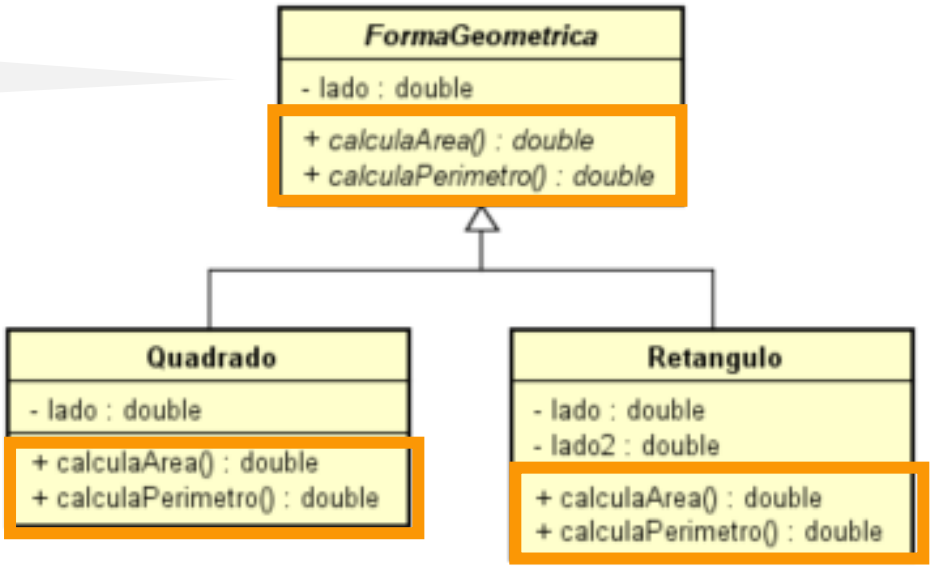
Subclasse de classe abstrata deve implementar **todos** os **métodos abstratos** da superclasse

# Método abstrato: sintaxe

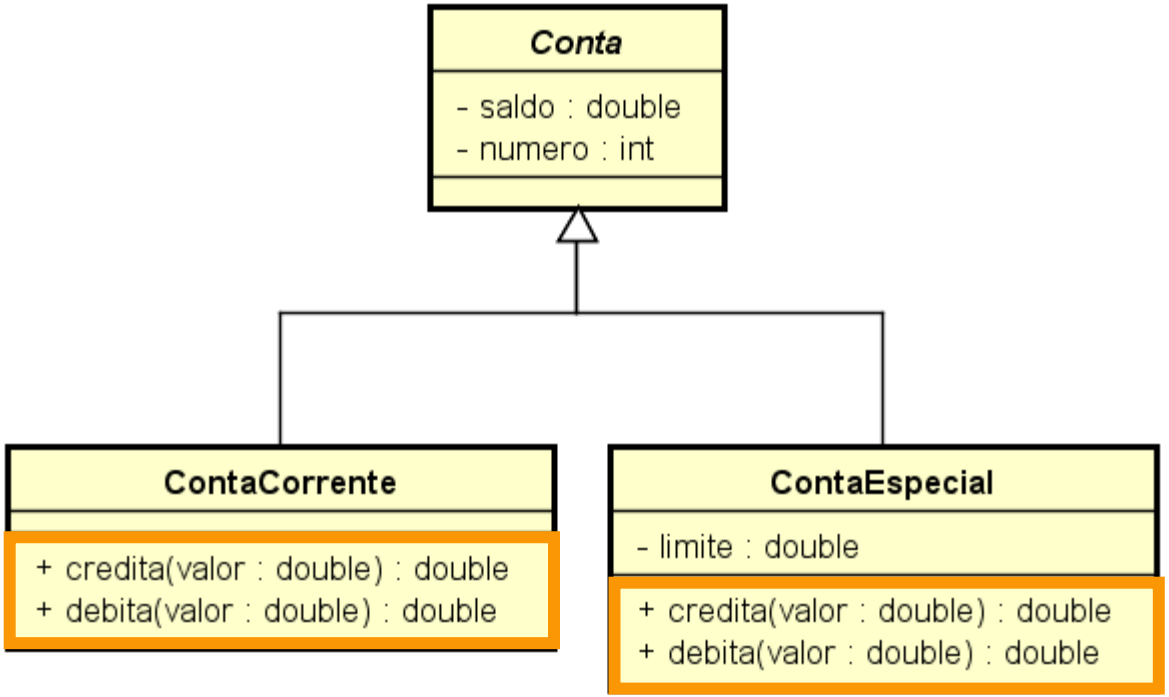
```
<modificador> abstract <tipo_retorno>
      <nome_metodo> (<lista_parâmetros>);
```

# Classe abstrata

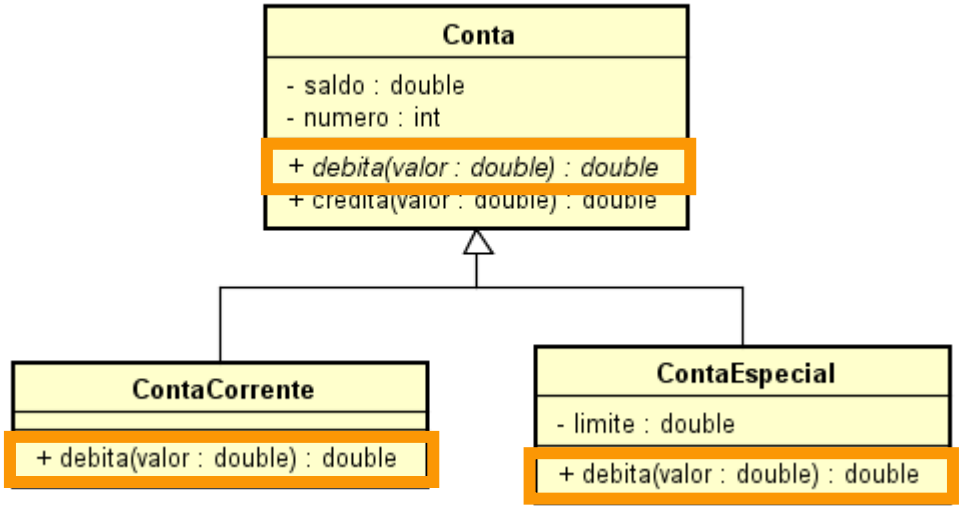
Qual método seria abstrato?



# Qual pode ser método abstrato?



# Qual pode ser Método abstrato?



# Método abstrato

```
public abstract class FormaGeometrica{

    private double lado;
    public abstract double calculaArea();
    public abstract double calculaPerimetro();
}
```

Um método abstrato não possui implementação, logo usa-se “;” para indicar o término da definição da assinatura do método

# Método abstrato

```
public class Quadrado extends FormaGeometrica{
    public double calculaArea(){
        return getLado()*getLado();
    }
    public double calculaPerimetro(){
        return 4*getLado();
    }

}
```

# Método abstrato

```
public class TesteAbstract{

 public static void main(…){

    FormaGeometrica f1 = new Quadrado(2);

    System.out.println(f1.calculaArea());

 }

}
```

# Resumo

| Modificador/Elemento | Classe | Método | Atributo |
|---|---|---|---|
| public | ✓ | ✓ | ✓ |
| private | x | ✓ | ✓ |
| protected | x | ✓ | ✓ |
| final | ✓ | ✓ | ✓ |
| static | ✓* | ✓ | ✓ |
| static final | x | x | ✓ |
| **abstract** | ✓ | ✓ | x |
| **abstract final/abstract** | x | x | x |
| **abstract private** | x | x | x |