Programação para Web II

Profa. Silvia Bertagnolli

Revisão Modificadores Java

UML e Orientação a Objetos

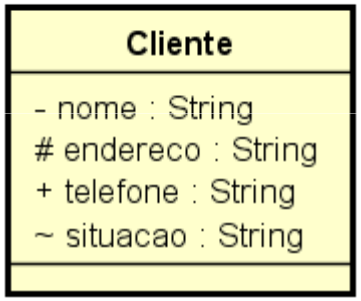
A UML (Unified Modeling Language) define um conjunto de diagramas relacionados com a orientação a objetos

O diagrama que mais se aproxima do código é o Diagrama de Classes, que possibilita definir regras de negócio nas associações entre as classes

Essas regras podem alterar o estado e/ou o comportamento das classes

Uma Classe na UML e uma classe na linguagem Java

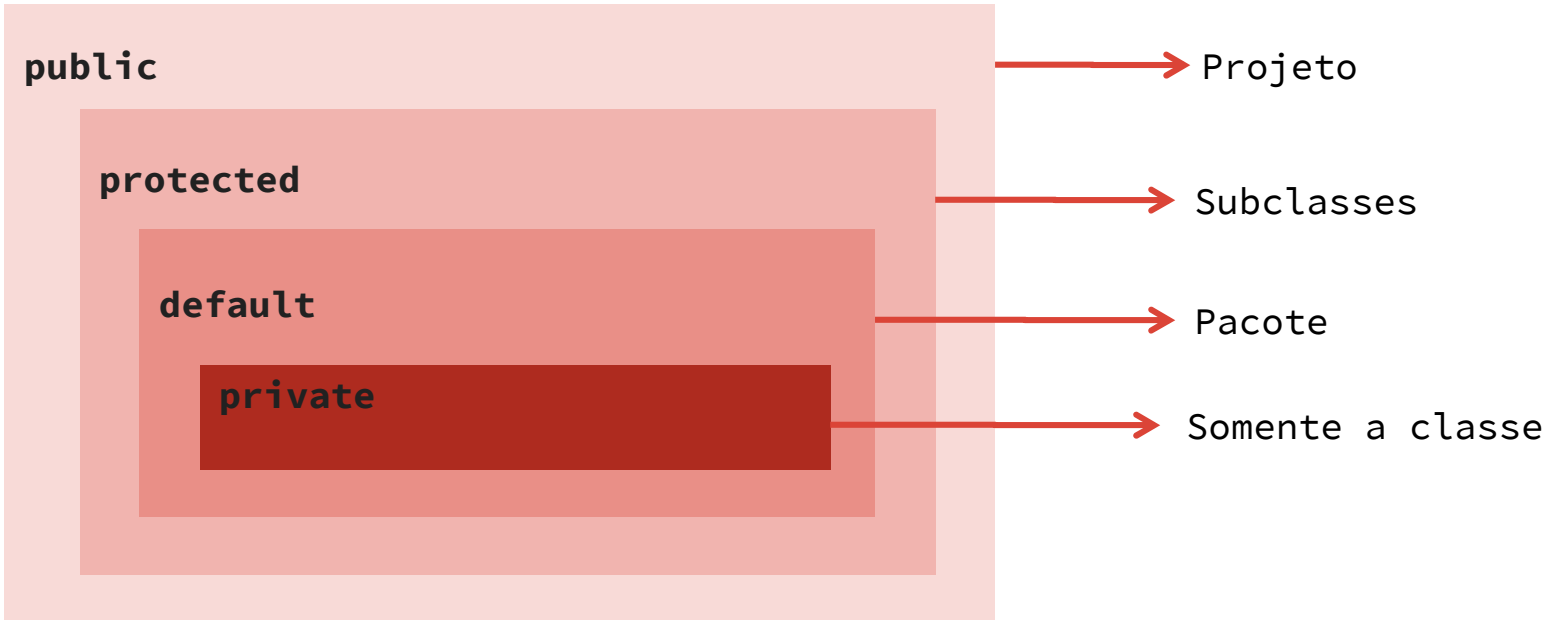
UML:



JAVA:

```
public class Cliente{

        private String nome;

         __________ String endereco;

        __________ String telefone;

        __________ String situacao;

...

}
```

Modificadores e sua Visibilidade



Recomendações

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

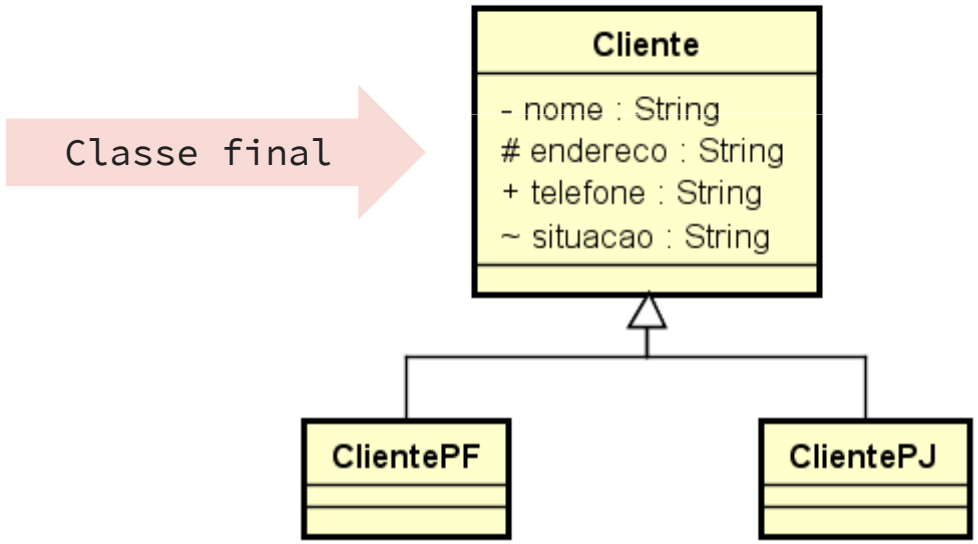
Resumo

Visibilidade	public	protected	private	default
Da mesma classe	✓	✓	✓	✓
De qualquer classe no mesmo pacote	✓	✓		✓
De qualquer classe que não seja uma subclasse externa ao pacote	✓			
De uma subclasse do mesmo pacote	✓	✓		✓
De uma subclasse externa ao pacote	✓	✓		

Modificador final

Classe final

Nenhuma outra classe jamais poderá estender esta classe



Classe final

Quando usar? Garantir que nenhum método da classe será sobreposto

Exemplo: java.lang.String

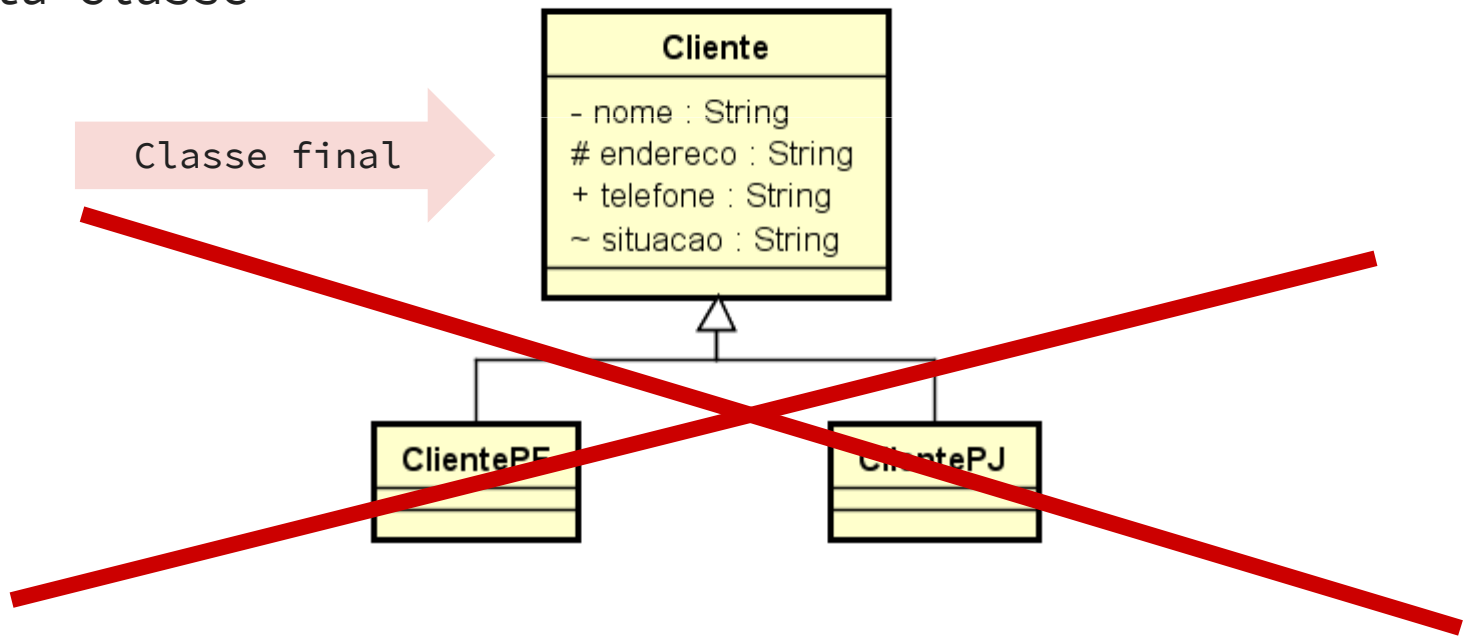
Vantagens:

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

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

Classe final (1)

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



Classe final

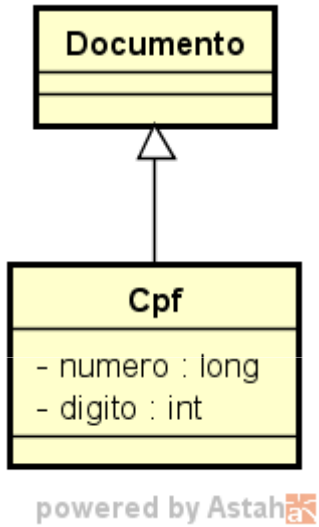
Sintaxe de declaração:

```
<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 TesteFinal1{

        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 de definição de um método final:

```
    <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**

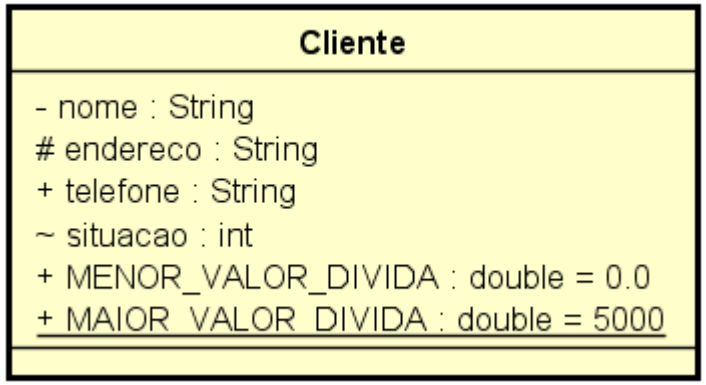
Sintaxe para declaração:

```
<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);

    }

}
```

Atributo final

A atribuição do valor da constante pode ser realizada de duas formas:

1. Na declaração do atributo
2. No(s) construtor(es) da classe (chamados “*blank FINAL variable*”)

A definição deve obrigatoriamente ocorrer em uma das duas formas possíveis

Se uma classe possuir vários métodos construtores, o atributo FINAL deverá ser inicializado em todos os construtores

Atributo final

Quando usado com os tipos primitivos
byte, short, int, long, char, float, double e boolean
permanecem com seus valores constantes

“Sua aplicação aos atributos que sejam objetos ou vetores também é permitida, no entanto, nesses casos, apenas a referência ao objeto ou ao vetor é fixa, ou seja, os valores dos atributos do objeto FINAL ou os valores contidos nas posições do vetor FINAL podem ser alterados, mas impede que sejam instanciados novamente.”

Revisando: encapsulamento

```
public class TesteEncapsulamento {
    public static void main(String args[]) {
        Cliente c = new Cliente();
        c.nome = "Fulano";
        c.endereco = "Rua X, 10";
        c.telefone = "33224455";
        c.situacao = Situacao.ATIVO;
        System.out.println(c.MENOR_VALOR_DIVIDA);
        System.out.println(Cliente.MAIOR_VALOR_DIVIDA);
    }
}
```

Variáveis e Parâmetros final

As variáveis e os parâmetros podem ser definidos como final

Variável final - Usada para assegurar que o valor não está sendo modificado indevidamente

```
final int num1 = in.nextInt();

final int num2 = in.nextInt();

if (num1 = num2) //gera erro
```

Variáveis e Parâmetros final

Parâmetro final - permite lembrar que parâmetros são passados por valor, não por referência

```
void troca(String s1, String s2) {
     String tmp = s1;
     s1 = s2;
     s2 = tmp;
}
-----------------------------------------
void troca(final String s1, final String s2) {
  String tmp = s1;
  s1 = s2; // ocorre erro de compilação aqui
  s2 = tmp; // ocorre erro de compilação aqui
}
```

Resumo

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

Modificador 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...) e o java.lang.Math.PI (3.14159...)

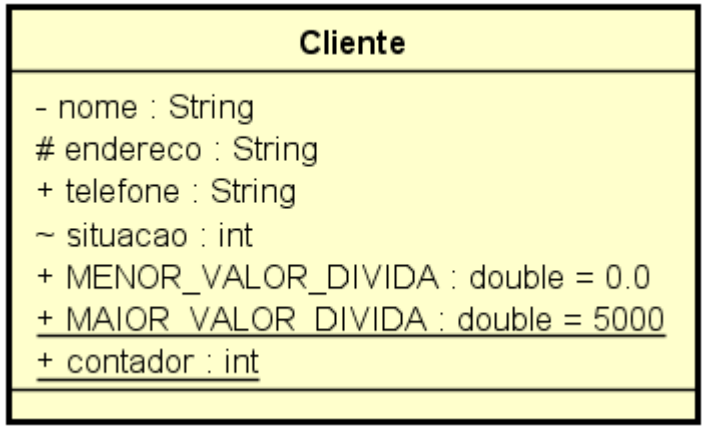
Sintaxe de definição:

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

ou:

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

Variável de Classe

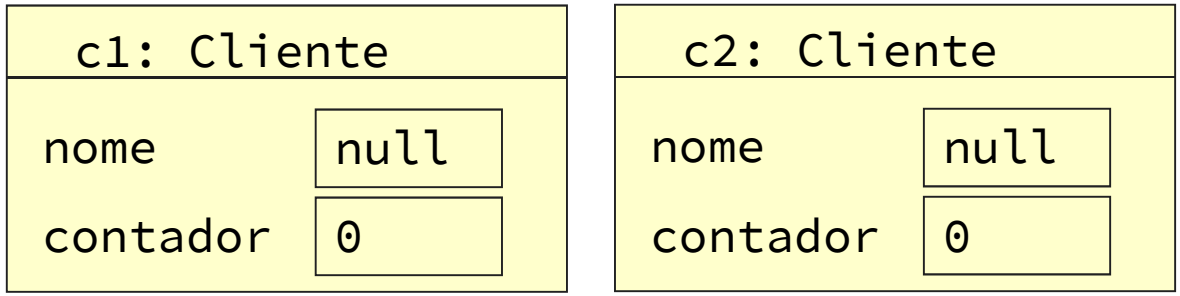



```
public class Cliente{
    //…
    private static int contador = 0;
    public static final double MAIOR_VALOR_DIVIDA = 5000.0;
   //…
}
```

Vamos adicionar o contador na classe Cliente

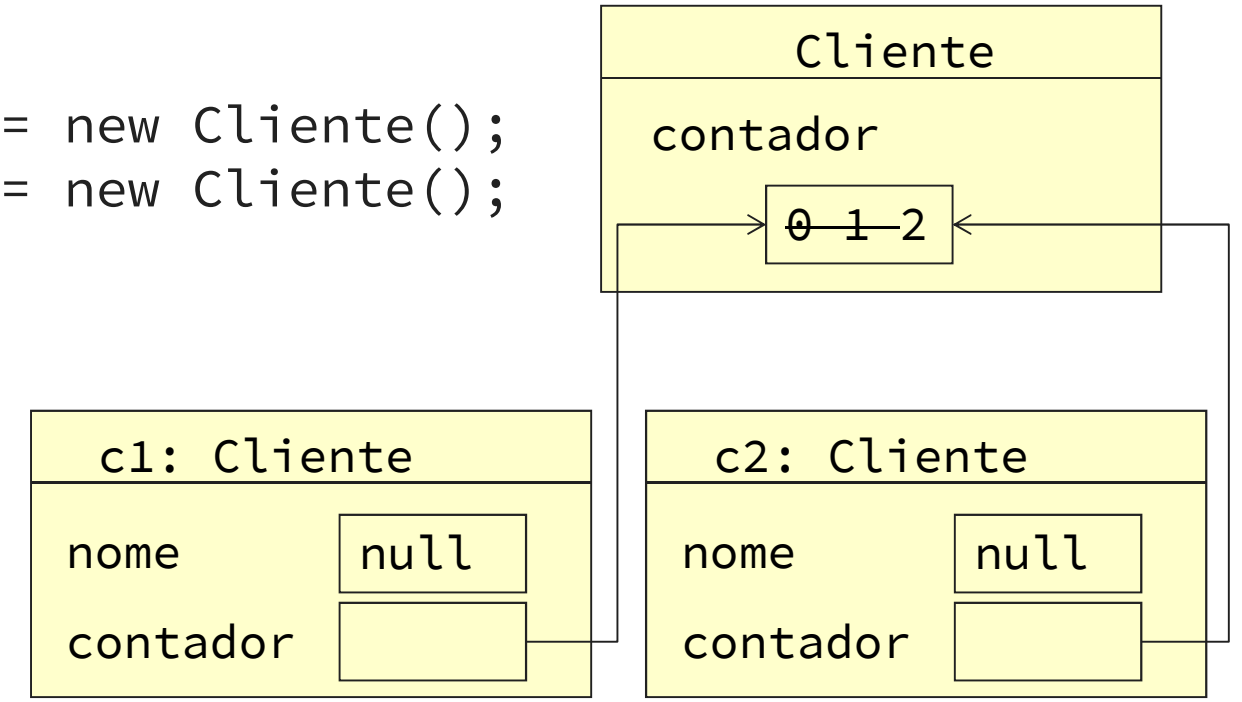
Se contador não fosse Variável de Classe

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



Se contador não fosse Variável de Classe

```
public class Teste6{
  ... main(...){
          Cliente c1 = new Cliente();
          Cliente c2 = new Cliente();
      }
}
```



Variável de Classe

```
public class Teste5{
  ... 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

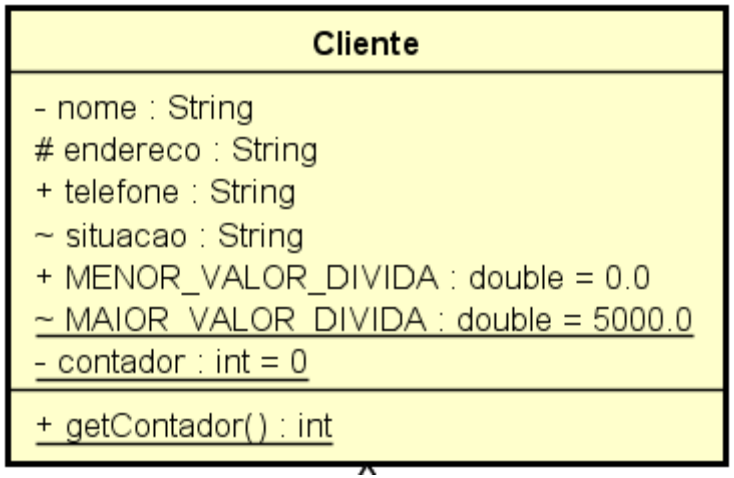
Método de Classe

Sintaxe de definição de um método estático:

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

Método de Classe

```
public class Cliente{
    //…
    private static int contador = 0;
    //…
    public static int getContador(){
        return contator;
   }
}
```



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();

Blocos de Inicialização Estáticos

Blocos de inicialização são trechos de código que serão executados automaticamente quando a classe for carregada em memória

Uso: desenvolvimento de rotinas de pré-configurações e validações, como a verificação da versão da JVM, da versão do sistema operacional, verificação se o acesso a algum dispositivo de rede está disponível, entre outras.

Blocos de Inicialização Estáticos

```
public class Estatica{
        public static String SO, versaojava;
        static {
            SO = System.getProperty("os.name");
            System.out.println(SO);
            versaojava = System.getProperty("java.version");
            System.out.println(versaojava);
            System.out.println(System.getProperties());
        }
//...
}
```

Classe Estática

O static pode ser usado para definição de classes aninhadas

```
public class ClasseExterna{
   public static class ClasseInterna{
        public void imprime() {
             System.out.println("método classe interna");
       }
   }
}
public class TesteStatic4 {
   public static void main(String[] args) {
      Externa.Interna interna = new Externa.Interna();
      interna.imprime();
   }
}
```

Classe Estática

O static pode ser usado para definição de classes aninhadas

```
public class ClasseExterna{
   public static class ClasseInterna{
        public void imprime() {
             System.out.println("método classe interna");
       }
   }
}
```

Classe Estática

Vantagens de classe interna:

1. Classe interna tem acesso aos atributos e métodos privados da classe externa, reduzindo a necessidade de getters e setters
2. Incentiva modularização
3. Coesão: a classe interna permite a definição do código junto à classe que irá utilizá-lo
4. Como classes internas geralmente não são relevantes para outras partes do programa, criá-las como classes privadas favorece o encapsulamento

Resumo

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

* Usado somente para classes internas

Modificador abstract

Classe abstrata

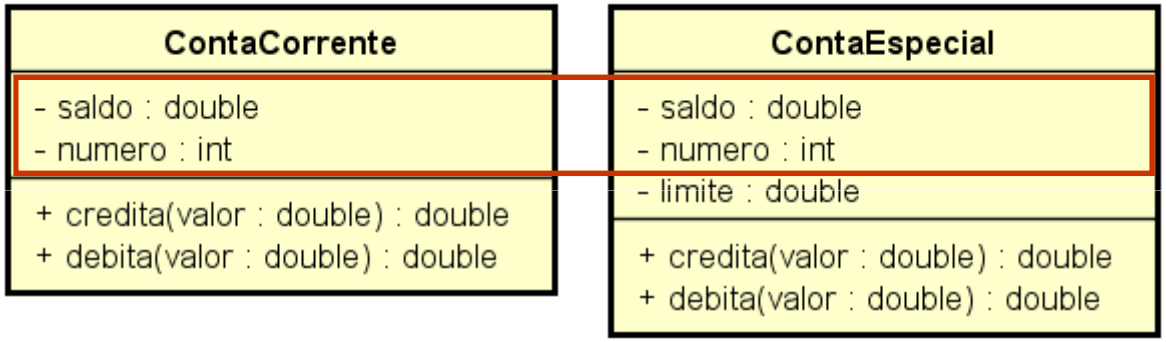
A única finalidade é ser estendida

Incompleta - geralmente, contém métodos abstratos

Métodos **podem** ser definidos nas subclasses

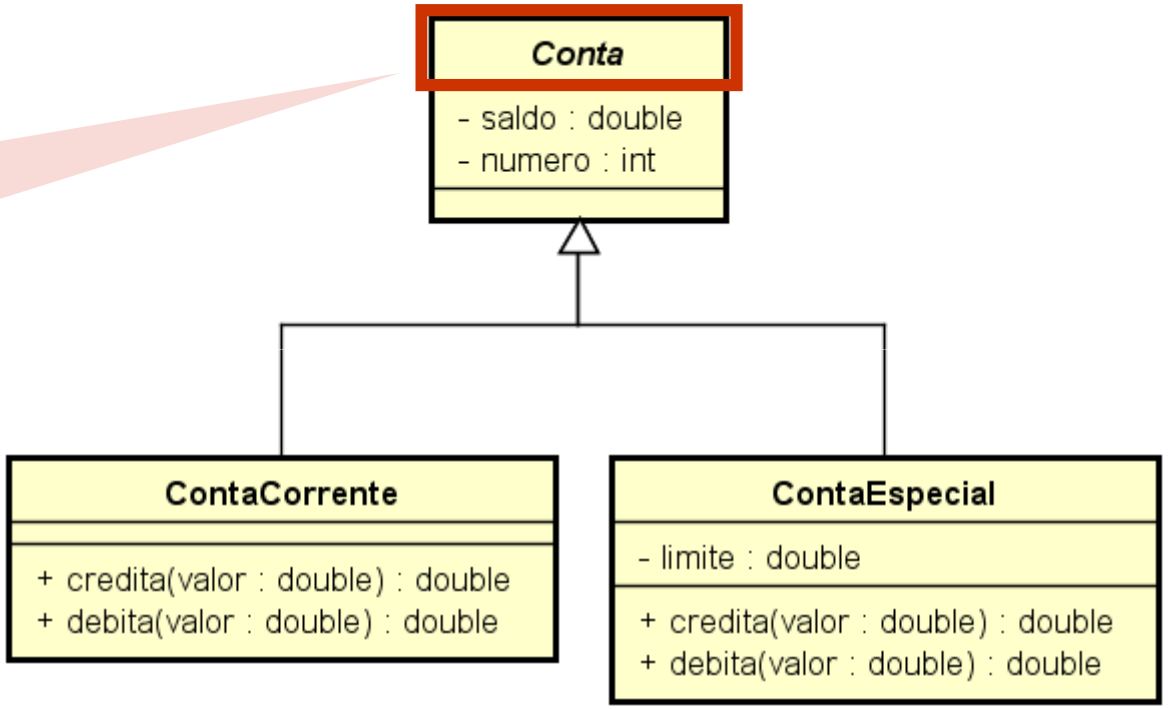
Obs.: se um método for definido como abstrato dentro de uma classe toda a classe deverá ser declarada como abstrata

Classe abstrata



Classe abstrata

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



Classe abstrata

Sintaxe de definição de uma classe abstrata:

```
<modificador> abstract class <nome_classe>{
        //...
}
```

Classe abstrata

```
public abstract class Conta{

    //…

    public Conta(){}

}
```

Classe abstrata

```
public class Teste7{

    public static void main(…){

        Conta c = new Conta();

        c.setSaldo(500.0);

    }

}
```

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

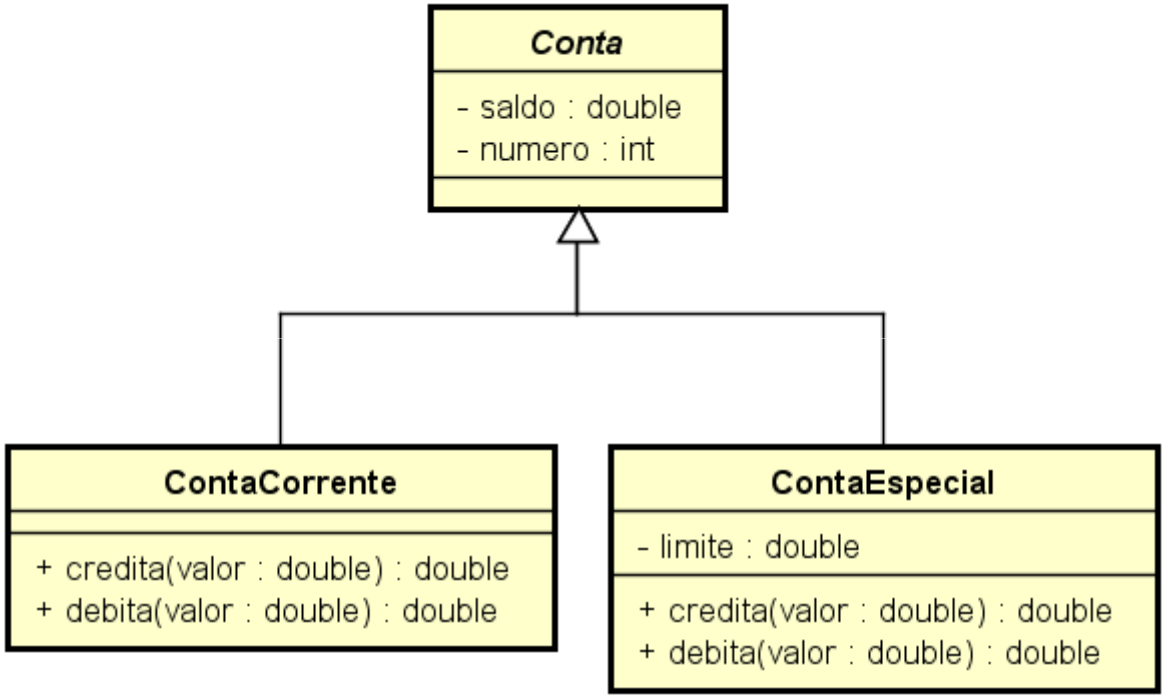
Subclasse de classe abstrata deve implementar **todos** os **métodos abstratos** da superclasse

Método abstrato

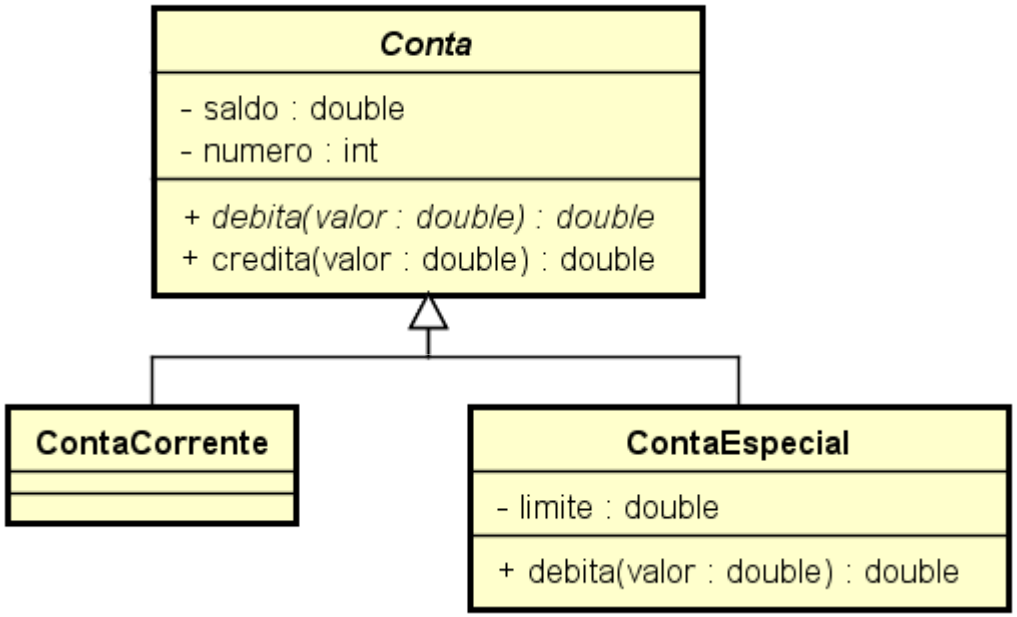
Sintaxe de definição de um método abstrato:

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

Qual pode ser Método abstrato?



Qual pode ser Método abstrato?



Método abstrato

```
public abstract class Conta{

    private double saldo;
    private int numero;
    public double credita(double valor){
        saldo += valor;
        return saldo;
    }
    public abstract double debita(double valor);

}
```

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

Método abstrato

```
public class ContaEspecial extends Conta{
    private double limite;
    public Conta(){}
    public double debita(double valor){
       if(getSaldo()+limite<=valor)
         setSaldo(getSaldo()-valor);
       return getSaldo();
    }
}
```

Método abstrato

```
public class Teste8{
 public static void main(…){
   ContaEspecial ce = new ContaEspecial(200.0);
   System.out.println(ce.debita(100.0));
 }
}
```

Resumo

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

Referências

http://docs.oracle.com/javase/specs/