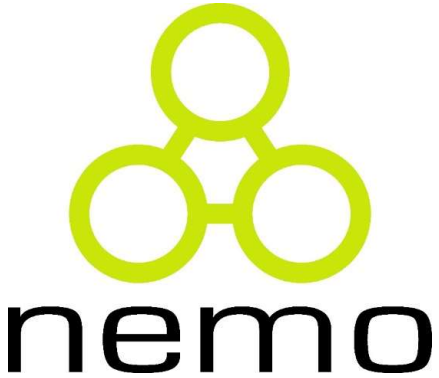


Programação III

Jordana S. Salamon

jssalamon@inf.ufes.br

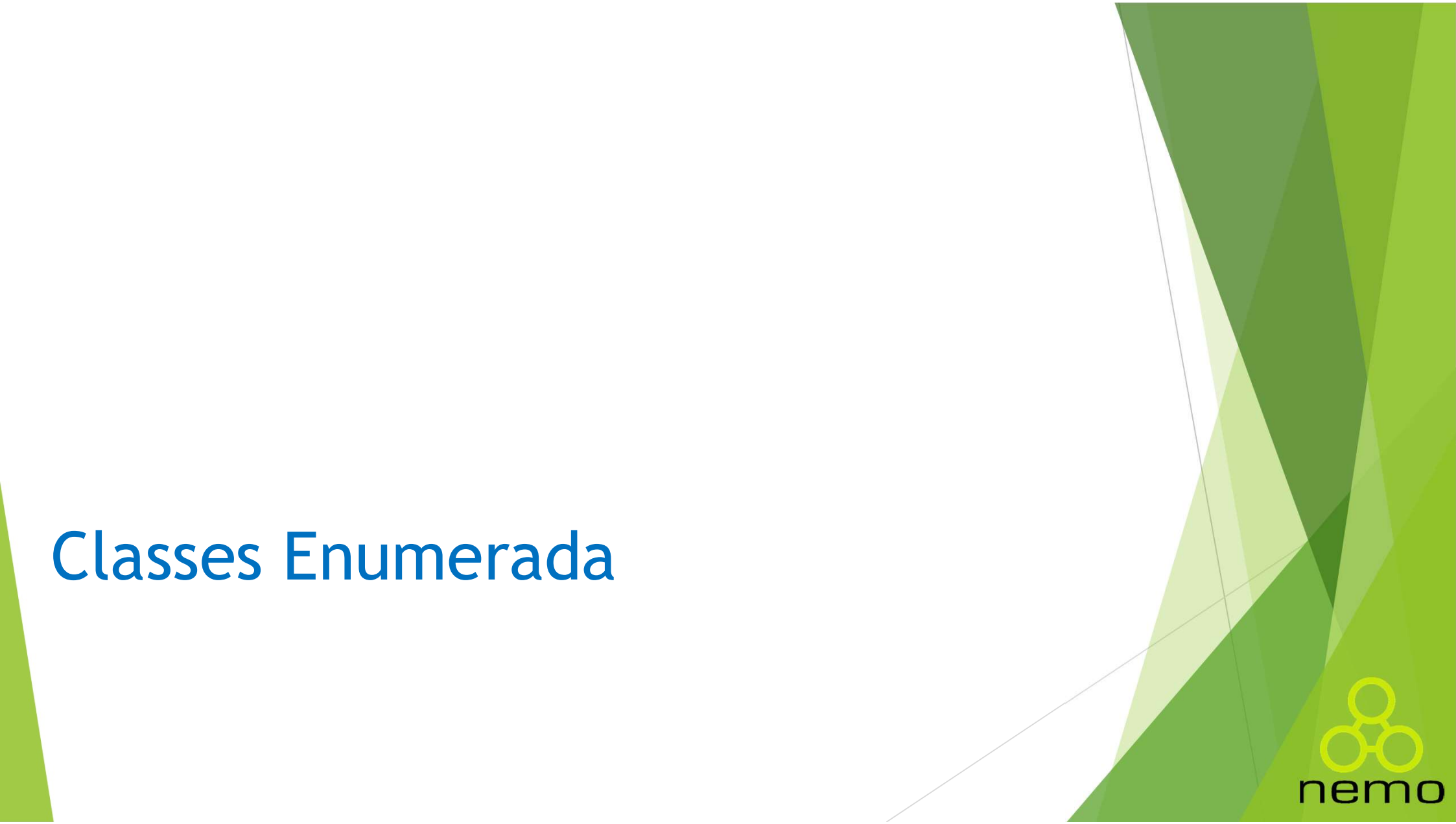
jordanasalamon@gmail.com

DEPARTAMENTO DE INFORMÁTICA

CENTRO TECNOLÓGICO

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Classes Enumerada



Classes Enumeradas

- Em Java é possível definir tipos enumerados que possuirão valores fixos. Ex:

```
public enum Genero {
    ACAO, AVENTURA, TERROR, SUSPENSE, COMEDIA, FICCAO;
}
```

- Para definir um valor especifico do Enum, basta utilizar o operador de acesso “.”

```
Genero g;
g = Genero.AVENTURA;
```

Classes Enumeradas

- Em Java é possível selecionar um Enum pelo nome (string), utilizando a função valueOf. Ex:

```
String nomeGenero;
System.out.println("\nDigite o nome do Genero do Jogo: ");
nomeGenero = s.next();
Genero g = Genero.valueOf(nomeGenero);
```

- A função transforma a string passada de parâmetro no enum específico.
- Contudo a string deve conter os caracteres da mesma forma que está representado no enum, senão gera erro. Ex:

```
g = Genero.valueOf("AVENTURA"); //Correto
g = Genero.valueOf("Aventura"); //Errado
```

Vetores e Listas em Java

Vetores (arrays) em Java

- Sintaxe herdada de C:

```
int[] filme1;   // Sintaxe preferida.
int filme2[];   // Sintaxe C...
```

- Acessar o vetor também é igual a C:

```
filme1[0] = 4;     // Como em C, índices começam em 0
filme1[1] = 8;  // .
filme1[2] = 15;    // .
filme1[3] = 16; // .
filme1[4] = 23;    // .
filme1[5] = 42; // e vão até (tamanho - 1)
```

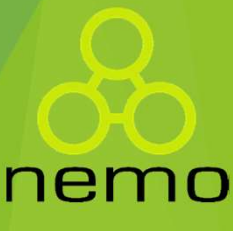
Generalizando o tamanho do vetor

- Vetores são criados **dinamicamente**:

```
// Quantos você quer?
Scanner scanner = new Scanner(System.in);
int tamanho = scanner.nextInt();
int[] vetor = new int[tamanho];
```

- E possuem o **atributo** `length` pra facilitar seu uso:

```
// Preenchendo o vetor...
for (int i = 0; i < vetor.length; i++)
  vetor[i] = i;
```



Listas Genericas

- Para a representação de listas genéricas em Java, assim como vector em C++, Java possui diversas implementação de listas na biblioteca java.util. A mais utilizada é a **ArrayList.**
- Contudo só é possível criar listas de objetos, ela não aceita tipos primitivos.
- Caso você queira criar uma lista de tipos primitivos, Java disponibiliza classes para representar cada tipo primitivo. Ex:
  - Interger
  - Float
  - Double
  - String

Listas Genericas

- A grande vantagem da ArrayList de Java em comparação com o vector é que ela possui métodos com um melhor controle da lista.

```
Jogo j1 = new Jogo("CS", 10.0);
Jogo j2 = new Jogo("PES", 20.0);
ArrayList<Jogo> l = new ArrayList(); //Cria a lista
l.add(j1); //Adiciona no final da lista
l.add(1, j2);//Adiciona na posição 1
if (l.contains(j1)){ // Vefica se um objeto está na lista
    System.out.println("Existe o nº 10 na lista");
}
if(l.isEmpty()){ //Verifica se a lista está vazia
    System.out.println("A lista está vazia");
}
int tam = l.size(); //Retorna a qtd de elementos da lista
l.remove(j1); //Remove o objeto do parametro, caso ele exista na lista
l.remove(0); //Remove o objeto da posição 0
l.clear(); //Limpa a lista
```

nemo

For Each

- A linguagem Java fornece um novo loop de repetição específico para listas de objeto, o **for each**.
- Basicamente ele percorre toda a lista, armazenando a referencia de cada objeto em um auxiliar a cada iteração do loop. Ex:

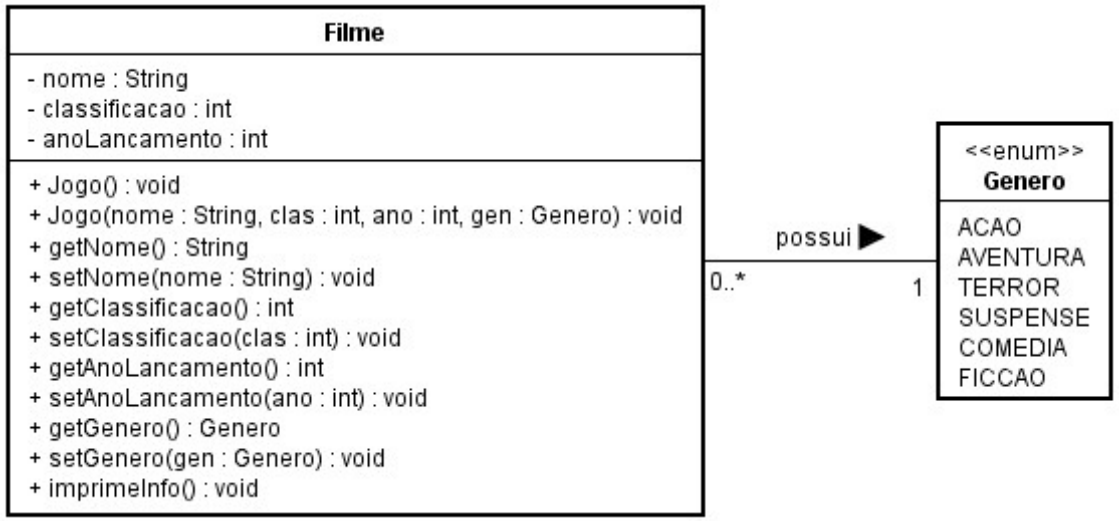
```
ArrayList<Jogo> jogos = new ArrayList();
for (Jogo jg : jogos) {
    jg.imprimeInfo();
}
```

- **jogos** é a minha lista e **jg** é um objeto auxiliar que vai receber cada posição da lista.

Exercício

Exercício

- Crie um projeto e implemente as seguintes classes:



- Na main, instancie um lista de filmes e desenvolva as funcionalidades de **cadastrar um novo Filme** (solicitando as informações do usuário) e de **imprimir a lista de filmes cadastrados**.

nemo

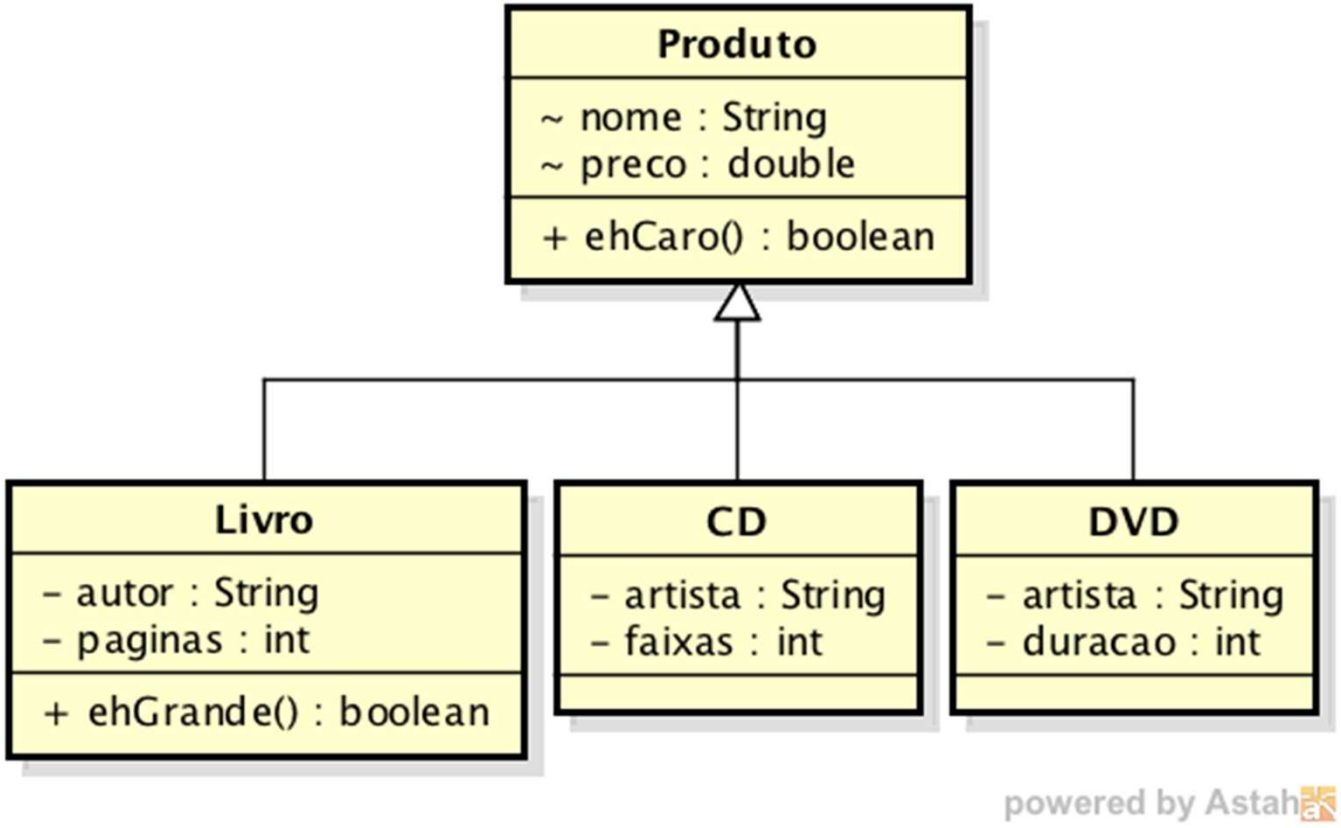
Herança em Java

História de Java

- Criação de **novas** classes **derivando** classes existentes;
- Relacionamento "**é um** [subtipo de]": um livro é um produto, um administrador é um usuário;
- Uso da palavra-chave `extends`;
- A palavra-chave é **sugestiva** – a classe que está sendo criada "**estende**" outra classe:
- **Partindo** do que já **existe** naquela classe...
- Pode **adicionar** novos recursos;
- Pode **redefinir** recursos existentes.

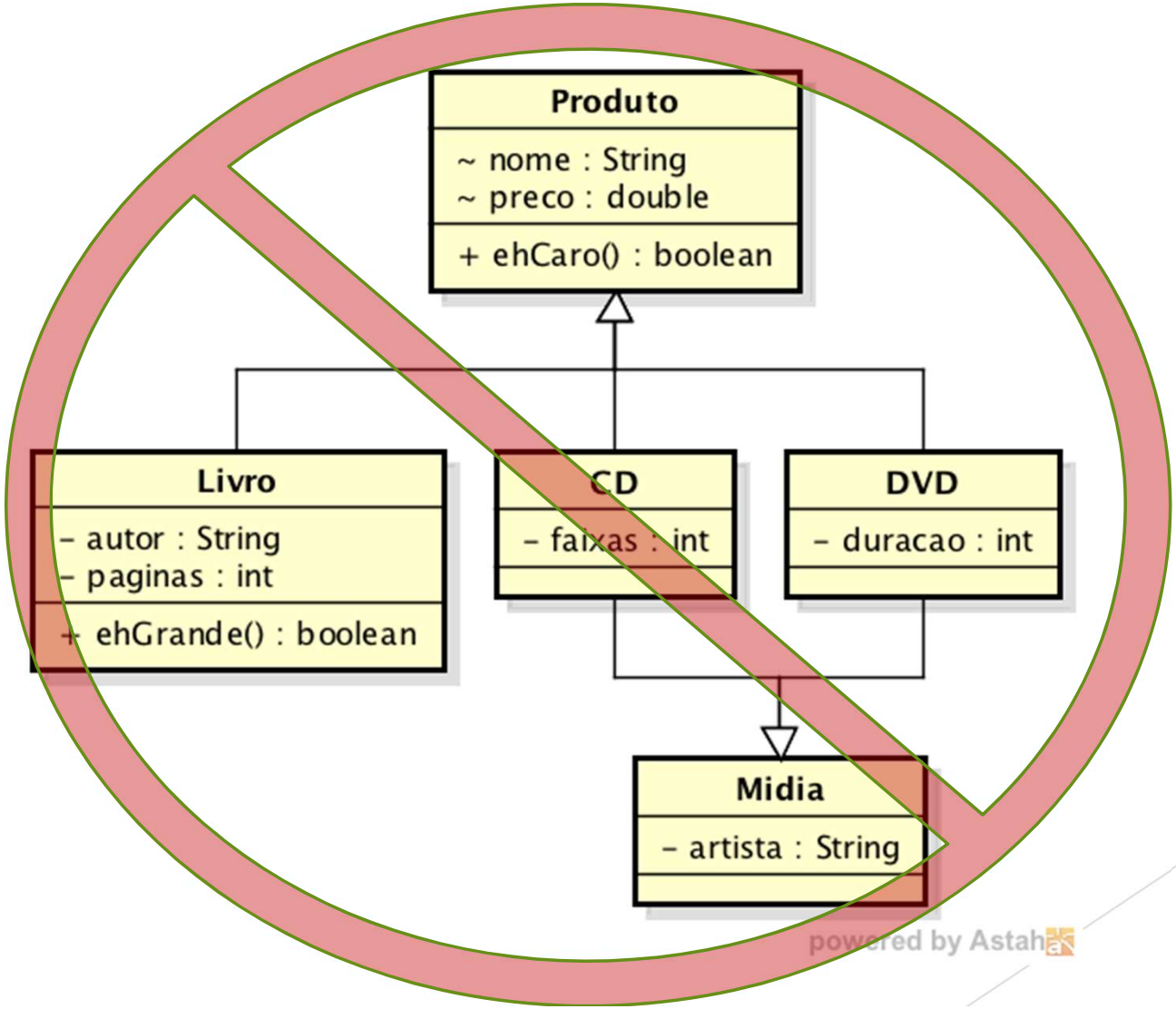
Java suporta herança simples

- Uma classe pode ter muitas subclases;
- Uma classe só pode ter uma superclasse.



nemo

Java não suporta herança múltipla



nemo

Sintaxe Herança

- Sintaxe:

```
class Subclasse extends Superclasse {
   /* ... */
}
```

- Semântica:
  - A subclasse herda todos os atributos e métodos que a superclasse possuir;
  - Subclasse é uma derivação, um subtipo, uma extensão da superclasse.

Chamando o método original

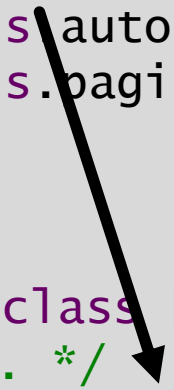
- Métodos **sobrescritos** podem chamar sua versão original na **superclasse** usando a palavra `super`:

```
class Produto {
  /* ... */
  public void imprimir() {
    System.out.println(nome + "," + preco);
  }
}

class Livro extends Produto {
  /* ... */
  public void imprimir() {
    super.imprimir();
    System.out.println(autor + "," + paginas);
  }
}
```

Chamando o construtor da superclasse

- É possível fazer o mesmo com **contrutores**:

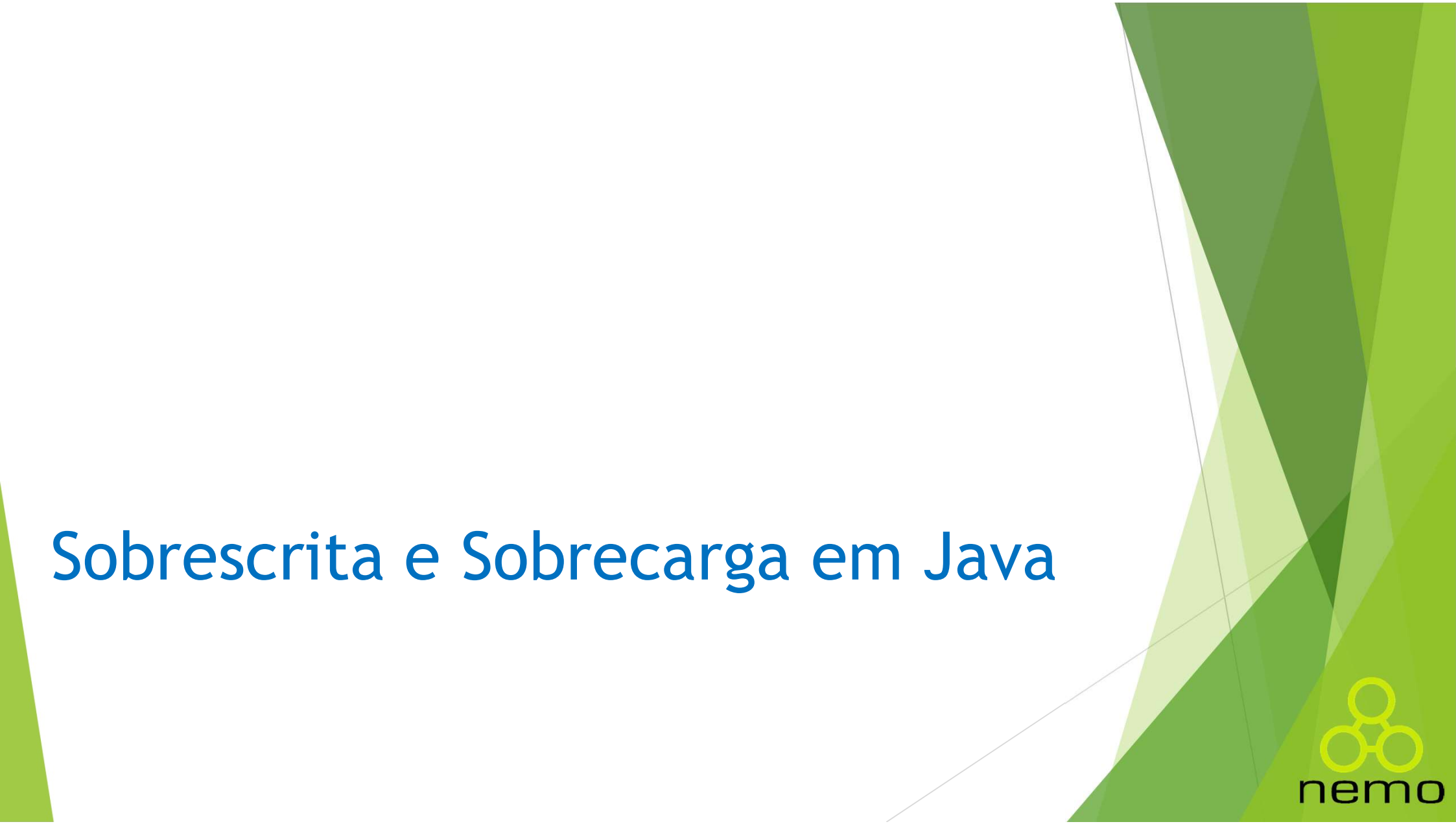
```
public class Livro extends Produto {
  /* ... */
  public Livro(String nome, double preco,
               String autor, int paginas) {
    super(nome, preco);
    this.autor = autor;
    this.paginas = paginas;
  }
}

public class Produto {
  /* ... */
  public Produto(String nome, double preco) {
    this.nome = nome;
    this.preco = preco;
  }
}
```



É tipo o `this()`, só que no `super()`...

nemo

Sobrescrita e Sobrecarga em Java



Sobrescrita

```
class Forma {
  public void desenhar() {
     // A substituir pela implementação oficial...
     System.out.println("Forma");
  }
}

class Circulo extends Forma {
  @Override
  public void desenhar() {
     System.out.println("Círculo");
  }
}

class Quadrado extends Forma { /* ... */ }

class Triangulo extends Forma { /* ... */ }
```

Anotação @Override

- Palavras precedidas de "@" são **anotações**:
  - **Meta-dados** úteis para o **compilador** ou algum outro componente da **plataforma** Java;
- @Override indica que o método **deve sobrescrever** um método herdado;
- Caso **contrário** (ex.: escrevemos o nome do método errado ou esquecemos um parâmetro), gera **erro** de **compilação**.

Sobrecarga

- Quando temos vários métodos com mesmo nome, dizemos que estamos **sobrecarregando** aquele nome;

- É útil para evitar redundâncias:
  - “lave o carro”, “lave a camisa”, “lave o cachorro”;
  - “laveCarro o carro”, “laveCamisa a camisa”, “laveCachorro o cachorro”.

- Fazemos isso quando definimos mais de um construtor para nossa classe!

- Podemos usar este conceito para qualquer método.

Sobrecarga

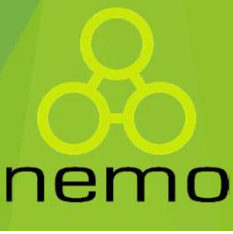
- Como Java distingue entre dois métodos com o mesmo nome?
- Pelos tipos dos parâmetros!

- Não podemos ter dois métodos com mesma assinatura, ou seja, mesmo nome e mesmos tipos de parâmetros;
- A ordem dos tipos de parâmetro influi:

```
/* OK! */
long multiplicar(long x, int y) { /* ... */ }
long multiplicar(int x, long y) { /* ... */ }
```

Sobrecarga

- Devemos ter cuidado ao usar sobrecarga em duas situações:
  - Tipos primitivos numéricos, que podem ser convertidos;
  - Classes que participam de uma hierarquia com polimorfismo.



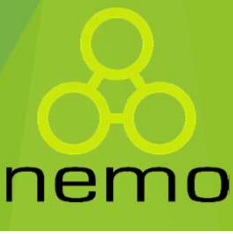
Final

A palavra reservada final

- Significa “Isto não pode ser mudado”;

- Dependendo do contexto, o efeito é levemente diferente;
- Pode ser usada em:
  - Dados (atributos / variáveis locais);
  - Métodos;
  - Classes.

- Objetivos:
  - Eficiência;
  - Garantir propriedades de projeto.



A palavra reservada final

- Constantes são comuns em LPs;
- Constantes conhecidas em tempo de compilação podem adiantar cálculos;
- Constantes inicializadas em tempo de execução garantem que o valor não irá mudar.
- Em Java, utiliza-se a palavra final:

```
public static final int MAX = 1000;
private final String NOME = "Java";
final double RAD = Math.PI / 180;
```

nemo

Referência constante

- Um primitivo constante nunca muda de valor;
- Uma referência constante nunca muda, mas o objeto pode mudar internamente:

```
public class Teste {
  public static final int MAX = 1000;
  private final Coordenada C = new Coordenada();
  public static void main(String[] args) {
    // Erro: MAX = 2000;
    // Erro: C = new Coordenada();
    C.x = 100;  // OK, se x for público!
  }
}
```

Dados finais não inicializados

```
class Viagem { }
class DadoFinalLivre {
  final int i = 0; // Final inicializado
  final int j;     // Final não inicializado
  final Viagem p;  // Referência final não inicializada

  // Finais DEVEM ser inicializados em
  // todos os construtores e somente neles
  DadoFinalLivre () {
     j = 1;
     p = new Viagem();
  }
  DadoFinalLivre (int x) {
     j = x;
     p = new Viagem();
  }
}
```

Argumentos finais

- Um parâmetro de um método pode ser final:
  - Dentro do método, funciona como constante.

```
public class Teste {
  public void soImprimir(final int i) {
    // Erro: i++;
    System.out.println(i);
  }
}
```

Métodos finais

- Métodos finais não podem ser sobrescritos por uma subclasse;
- Chamada do método inline (maior eficiência).

```
class Telefone {
  public final void discar() { }
}

// Não compila: discar() é final!
class TelefoneCelular extends Telefone {
  public void discar() { }
}
```

Métodos finais

- Métodos privativos não podem ser acessados;
- Portanto, são finais por natureza (as subclasses não têm acesso a eles).

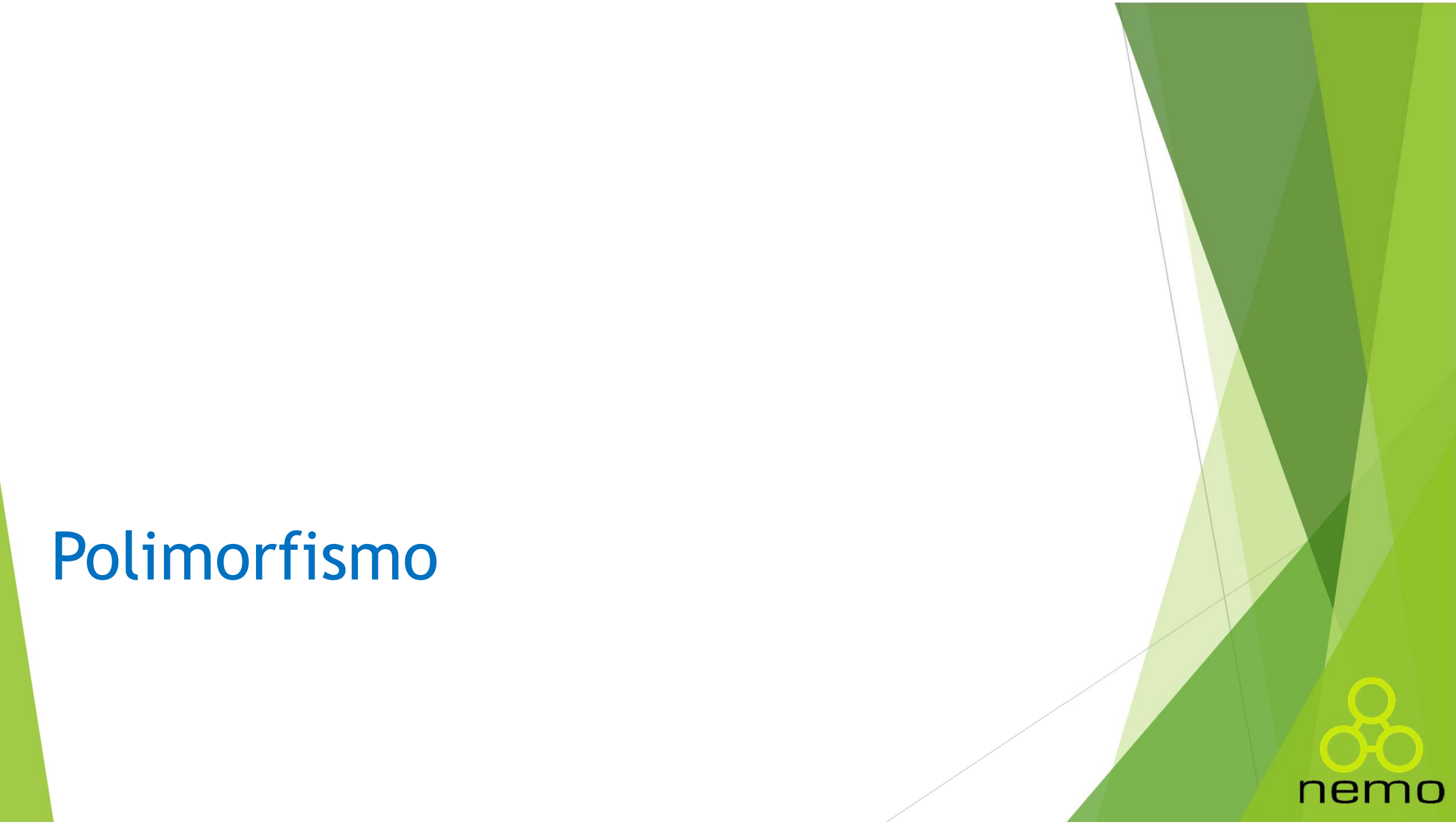
```
class Telefone {
  private final void checarRede() { }
}

// OK. São dois métodos diferentes!
class TelefoneCelular extends Telefone {
  private final void checarRede() { }
}
```

Classes finais

- Classes finais não podem ter subclasses;
- Por consequência, todos os métodos de uma classe final são automaticamente finais.

```
class Telefone { }

final class TelefoneCelular extends Telefone { }

// Erro: TelefoneCelular é final!
class TelefoneQuantico extends TelefoneCelular { }
```

Polimorfismo



Polimorfismo

- Do grego poli + morphos = múltiplas formas;
- Característica OO na qual se admite tratamento idêntico para objetos diferentes baseado em relações de semelhança;
- Em outras palavras, onde uma classe base é esperada, aceita-se qualquer uma de suas subclasses.

Polimorfismo

```
class Forma {
  public void desenhar() {
     // A substituir pela implementação oficial...
     System.out.println("Forma");
  }
}

class Circulo extends Forma {
  @Override
  public void desenhar() {
     System.out.println("Círculo");
  }
}

class Quadrado extends Forma { /* ... */ }

class Triangulo extends Forma { /* ... */ }
```

Polimorfismo

- Duas questões sobre o método desenhar():
  - Ele tem que existir pra todos;
  - Ele tem que fazer algo diferente para cada forma!

```
public class AplicativoDesenho {
  private static void desenhar(Forma[] fs) {
     for (int i = 0; i < fs.length; i++)
        fs[i].desenhar();
  }
  public static void main(String[] args) {
     Forma[] formas = new Forma[] {
        new Circulo(), new Forma(),
        new Quadrado(), new Triangulo()
     };
     desenhar(formas);
  }
}
```

Ampliação (upcasting)

- Ampliação (upcasting) é a conversão implícita de uma subclasse para uma superclasse:

```
public class AplicativoDesenhoSimples {
  public static void desenhar(Forma f) {
    f.desenhar();
  }

  public static void main(String[] args) {
    Circulo c = new Circulo();
    desenhar(c);                  // Upcasting!
    Forma f = new Quadrado();   // Upcasting!
  }
}
```

Ampliação (upcasting)

- O compilador realmente não sabe qual é o tipo do objeto.
- Quando realizamos ampliação, “esquecemos” o tipo de um objeto:
- Forma f = new Quadrado();
- Não sabemos mais qual é a classe específica de f.
- Sabemos apenas que ele é uma forma;
- Por que fazer isso?

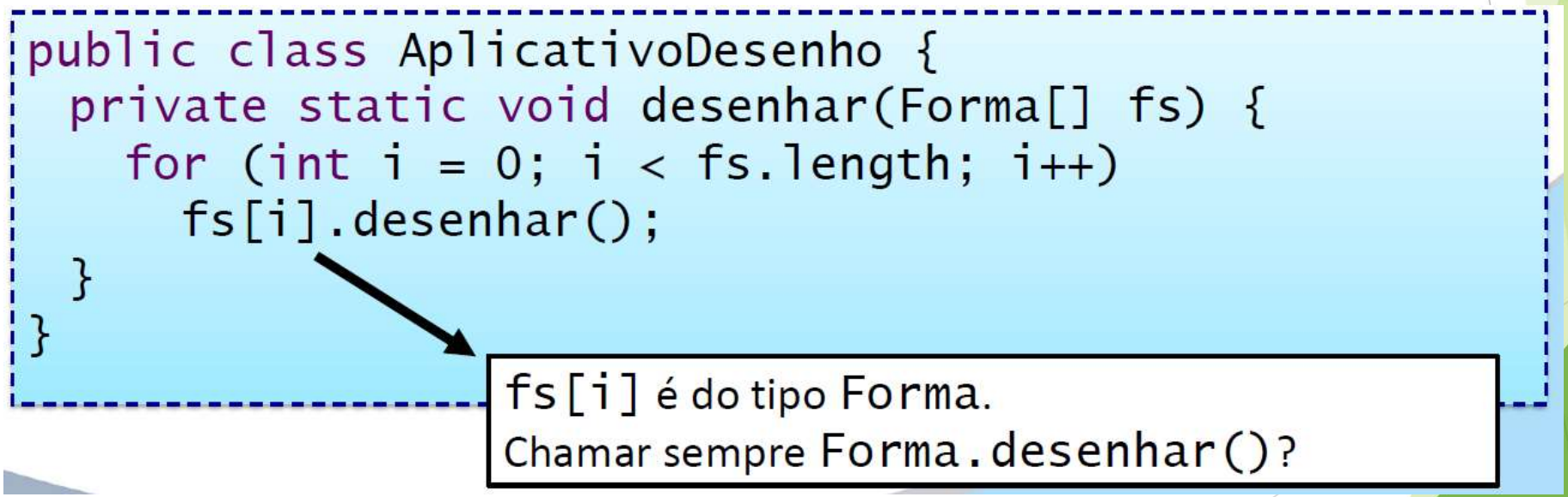
Ampliação (upcasting)

- Fazemos ampliação para escrevermos métodos mais gerais, para poupar tempo e esforço:

```
class AplicativoDesenhoTosco {
  public static void desenhar(Circulo c) {
    c.desenhar();
  }

  public static void desenhar(Quadrado q) {
    q.desenhar();
  }

  public static void desenhar(Triangulo t) {
    t.desenhar();
  }
}
```

Amarração

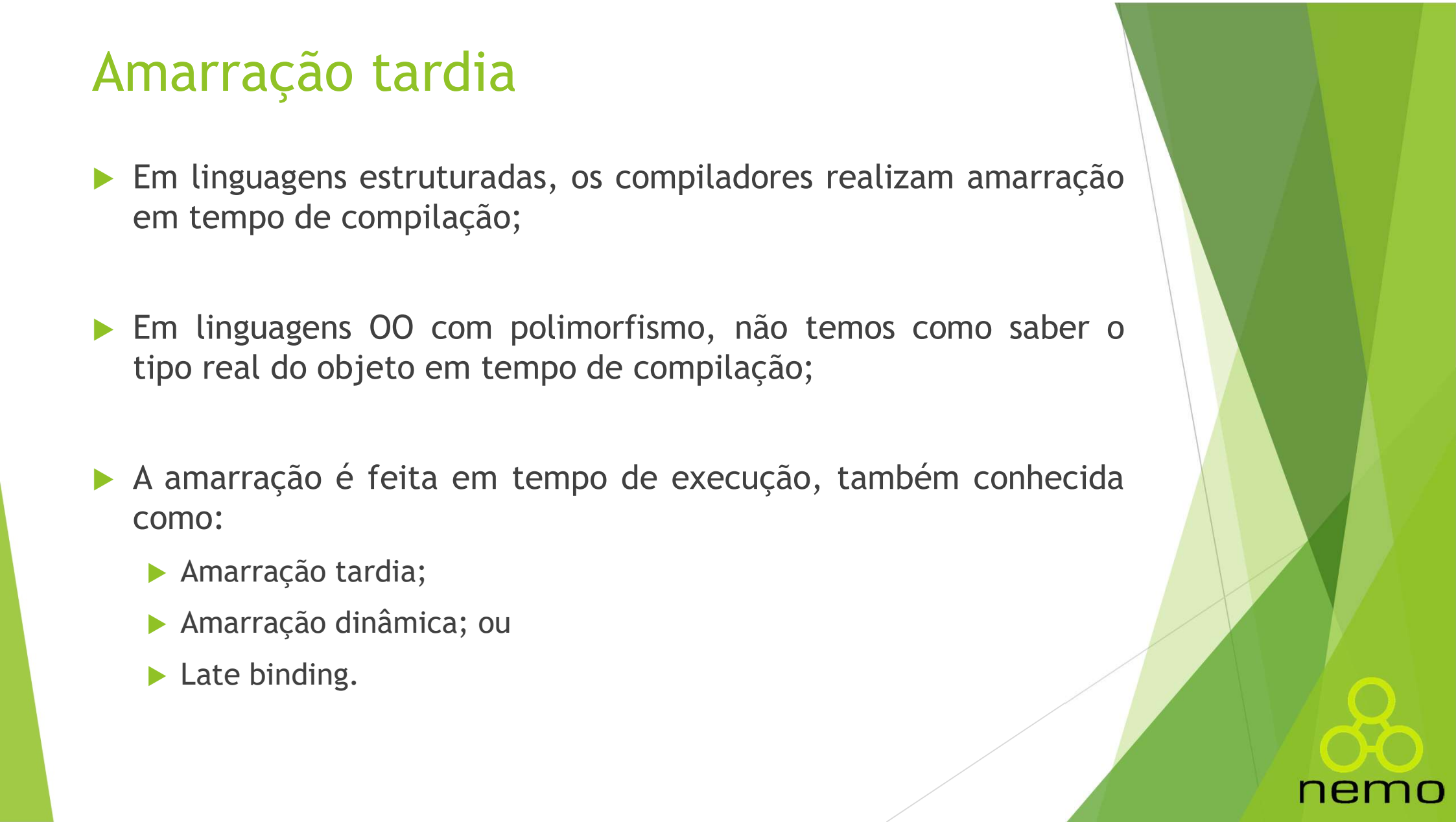
- No entanto, se trabalhamos com Forma, como saber qual implementação executar quando chamamos um método?



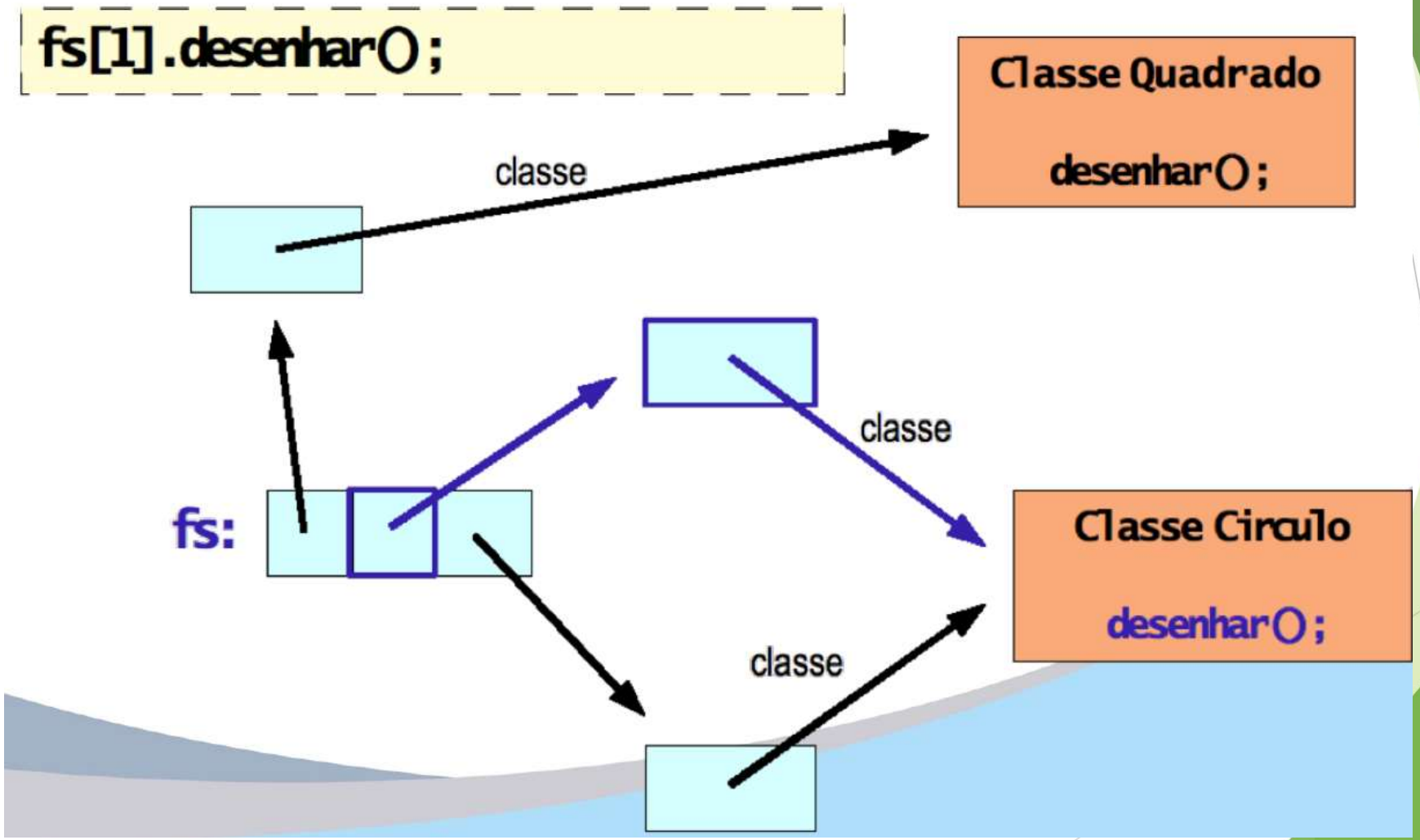
nemo

Amarração tardia

- Em linguagens estruturadas, os compiladores realizam amarração em tempo de compilação;
- Em linguagens OO com polimorfismo, não temos como saber o tipo real do objeto em tempo de compilação;
- A amarração é feita em tempo de execução, também conhecida como:
  - Amarração tardia;
  - Amarração dinâmica; ou
  - Late binding.



Amarração tardia



nemo

Amarração tardia

- Quando usar:
- Amarração dinâmica é menos eficiente;
- No entanto, ela que permite o polimorfismo;
- Java usa sempre amarração dinâmica;
- A exceção: se um método é final, Java usa amarração estática (pois ele não pode ser sobrescrito);
- Você não pode escolher quando usar um ou outro. É importante apenas entender o que acontece.

A Classe Object

A classe Object

- Em Java, todos os objetos participam de uma mesma hierarquia, com uma raiz única.
- Esta raiz é a classe java.lang.Object

```
class Filme { }

/* É equivalente a: */

class Filme extends Object { }
```

A classe Object

- A vantagem disso é que a classe Object possui alguns métodos úteis:
  - clone(): cria uma **cópia** do objeto (uso avançado);
  - equals(Object o): verifica se objetos são **iguais**;
  - finalize(): chamado pelo **GC** (não é garantia);
  - getClass(): retorna a **classe** do objeto;
  - notify(), notifyAll() e wait(): para uso com **threads**;
  - toString(): **converte** o objeto para uma representação como String.

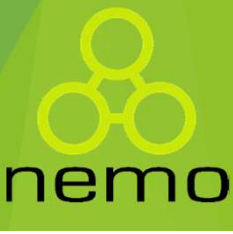
O método toString()

- toString() é chamado sempre que:
  - Tentamos **imprimir** um objeto;
  - Tentamos **concatená-lo** com uma string.

```
public class Loja {
  public static void main(String[] args) {
     Produto p = new Produto("CD", 30.0);
     System.out.println(p);
  }
}

// Resultado (toString() herdado de Object):
// Produto@10b62c9
```

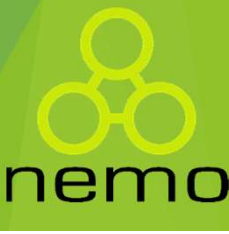
O método toString()

```
class Produto {
  /* ... */
  @Override
  public String toString() {
     return nome + " (R$ " + preco + ")";
  }
}
public class Loja {
  public static void main(String[] args) {
     Produto p = new Produto("CD", 30.0);
     System.out.println(p);
  }
}

// Resultado (toString() sobrescrito):
// CD (R$ 30.0)
```



O método equals()

```
class Valor {
  int i;
  public Valor(int i) { this.i = i; }
}

public class Teste {
  public static void main(String[] args) {
     int m = 100;
     int n = 100;
     System.out.println(m == n);          // true

     Valor v = new Valor(100);
     Valor u = new Valor(100);
     System.out.println(v == u);          // false
  }
}
```



O método equals()

```
class Valor {
  int i;
  public Valor(int i) { this.i = i; }

  @Override
  public boolean equals(Object o) {
     return (o instanceof Valor)
              && (((Valor)o).i == i);
  }
}

public class Teste {
  public static void main(String[] args) {
     Valor v = new Valor(100);
     Valor u = new Valor(100);
     System.out.println(v.equals(u));    // true
  }
}
```

That’s all Folks!