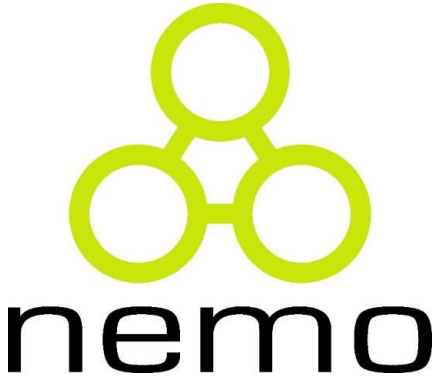


Programação III

Jordana S. Salamon

jssalamon@inf.ufes.br

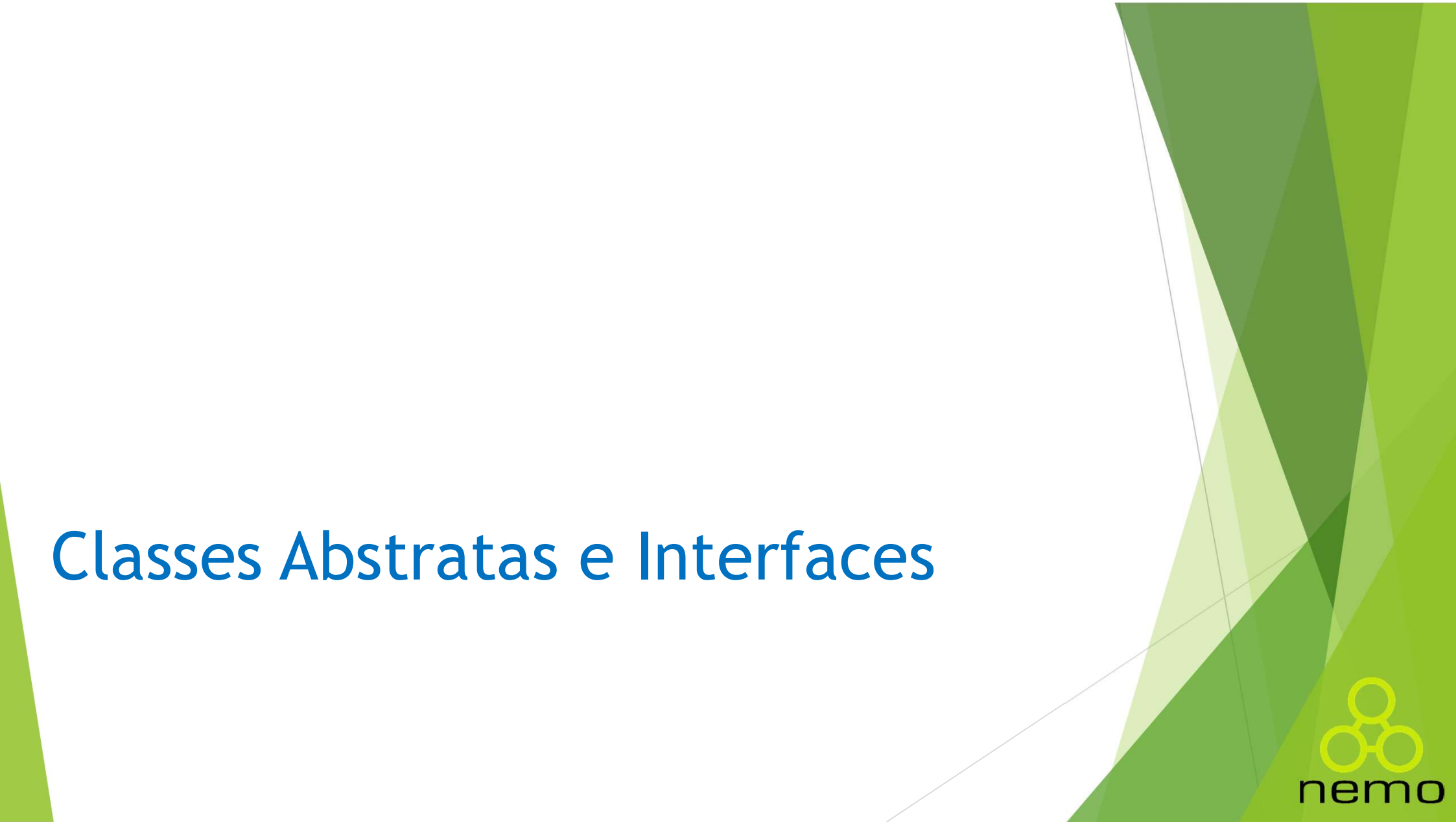
jordanasalamon@gmail.com

DEPARTAMENTO DE INFORMÁTICA

CENTRO TECNOLÓGICO

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Classes Abstratas e Interfaces



Classes e métodos abstratos

- Uma classe que possui métodos abstratos deve ser declarada como abstrata:

```
abstract class Forma {
  public abstract void desenhar();
}

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

Classes abstratas

- **Não permitem** criação de **instâncias** (objetos):
  - Um método abstrato **não possui** implementação, portanto não pode ser chamado.
- Para ser útil, deve ser **estendida**:
  - Suas **subclasses** devem **implementar** o método ou declararem-se como abstratas.
- Servem para definir **interfaces** e prover algumas **implementações** comuns.

Classes e métodos abstratos

Métodos estáticos podem ser abstratos?	Não
Construtores podem ser abstratos?	Não
Classes abstratas podem ter construtores?	Sim

Lembre-se: construtores são chamados pelas subclasses!

Métodos abstratos podem ser privativos?	Não
Uma classe abstrata podem estender uma normal?	Sim
Posso ter uma classe abstrata sem nenhum método abstrato?	Sim

nemo

Classes abstratas (puras) e concretas

```
// Classe abstrata pura.
abstract class Forma {
  public abstract void desenhar();
  public abstract void aumentar(int t);
}

// Classe abstrata.
abstract class Poligono extends Forma {
  private int lados;
  public Poligono(int lados) {
    this.lados = lados;
  }
  public int getLados() { return lados; }
  public abstract void pintar(int cor);
}
```

Classes abstratas (puras) e concretas

```
// Classe concreta.
class Retangulo extends Poligono {
  public Retangulo() {
    super(4);
  }
  @Override public void desenhar() {
    System.out.println("Retangulo.desenhar");
  }
  @Override public void aumentar(int t) {
    System.out.println("Retangulo.aumentar");
  }
  @Override public void pintar(int cor) {
    System.out.println("Retangulo.pintar");
  }
}
```

Interfaces

- Uma classe **abstrata** é **pura** quando:
  - Possui métodos **abstratos**;
  - Não possui métodos **concretos**;
  - Não possui **atributos** (não-static).
- Java **oferece** a palavra reservada `interface`:
  - Cria uma classe **abstrata pura**;
  - Chamaremos pelo nome de **interface**;
  - Ao conversar com outros programadores, **cuidado** para não **confundir** com “interface com o usuário”.
- Exemplos de Interfaces padrões da linguagem: **comparable**, **serializable**.

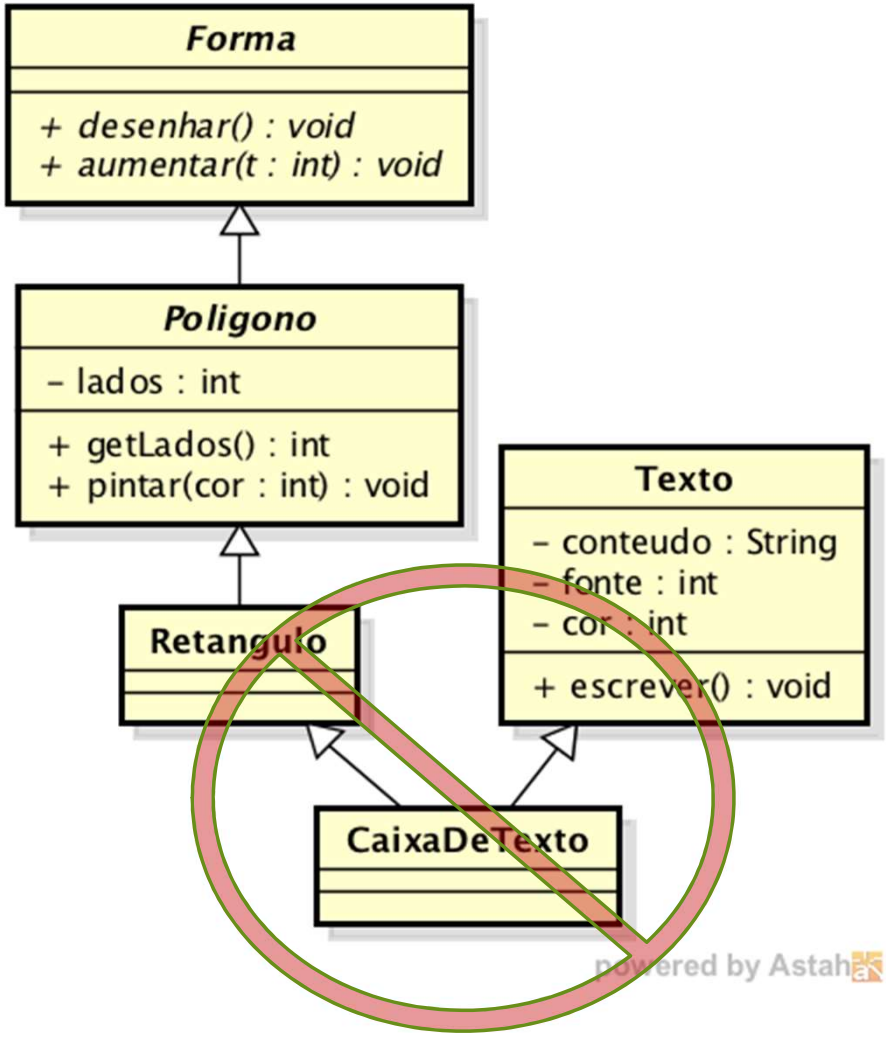
Interfaces

```
interface Forma {
  void desenhar();
  void aumentar(int t);
}

abstract class Poligono implements Forma {
  private int lados;

  public Poligono(int lados) {
     this.lados = lados;
  }

  public int getLados() { return lados; }

  public abstract void pintar(int cor);
}
```

Interfaces

```
class Linha implements Forma {
  private double x1, y1, x2, y2;

  @Override
  public void desenhar() {
     /* ... */
  }

  @Override
  public void aumentar(int t) {
     /* ... */
  }
}
```

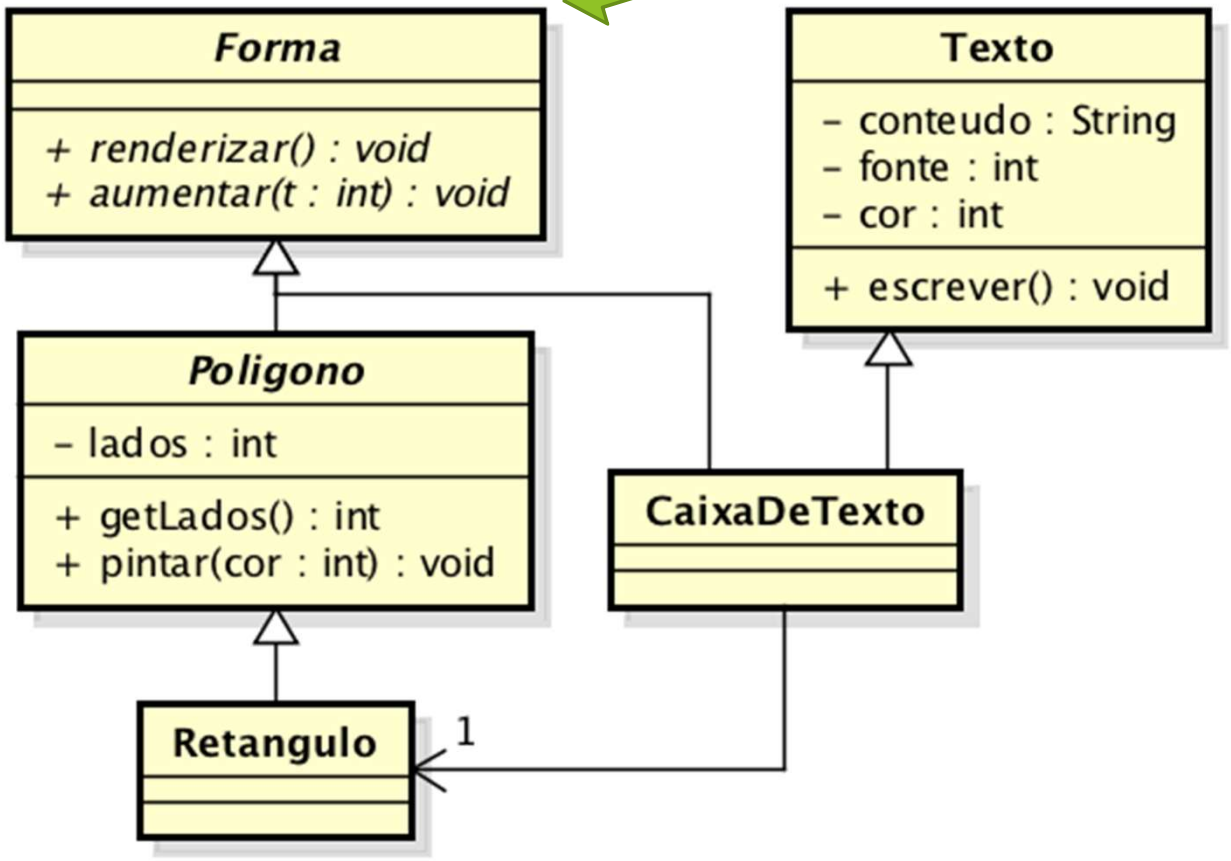
Herança Múltipla em Java



nemo

Herança Múltipla em Java

Lembrando: Forma é interface!



nemo

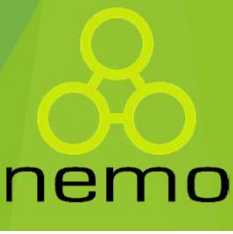
Herança Múltipla em Java

```
class CaixaDeTexto extends Texto implements Forma {
  private Retangulo caixa;

  /* ... */

  public CaixaDeTexto() {
     // Parâmetros foram omitidos para simplificar...
     caixa = new Retangulo();
  }

  public void renderizar() {
     // Desenha a caixa.
     caixa.renderizar();

     // Escreve o texto.
     escrever();
  }
}
```

O mecanismo de RTTI

Polimorfismo e extensão

- Com polimorfismo, podemos esquecer a classe de um objeto e trabalhar com a superclasse:
  - A interface de ambas é a mesma;
  - A amarração dinâmica garante que o método da classe correta será executado.
- O que acontece se a subclasse estende a superclasse (adiciona mais funcionalidade)?
- Se a superclasse não possui aquela funcionalidade, não podemos chamá-la!



Polimorfismo e extensão

```
interface Animal {
  void comer();
}

class Cachorro implements Animal {
  @Override public void comer() {
    System.out.println("Comendo um osso...");
  }

  public void latir() {
    System.out.println("Au Au!");
  }
}
```

Polimorfismo e extensão

```
class Gato implements Animal {
  @Override public void comer() {
    System.out.println("Comendo um peixe...");
  }

  public void miar() {
    System.out.println("Miau!");
  }
}
```

Polimorfismo e extensão

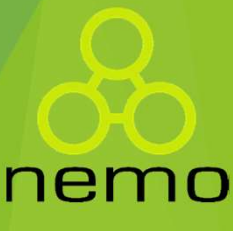
```
public class Teste {
  public static void main(String[] args) {
    Animal[] vet = new Animal[] {
        new Cachorro(), new Gato(),
        new Gato(), new Cachorro()
    };

    for (int i = 0; i < vet.length; i++) {
      vet[i].comer();
      // Erro: vet[i].latir();
    }
  }
}
```

#comofas?

Estreitamento (*downcast*)

- Precisamos relembrar a classe específica do objeto para chamarmos métodos que não estão na interface da superclasse;
- Para isso faremos estreitamento:

Ampliação (upcast)	Estreitamento (downcast)
`int` para `long`	`long` para `int`
`float` para `double`	`double` para `float`
`Cachorro` para `Animal`	`Animal` para `Cachorro`
`Gato` para `Animal`	`Animal` para `Gato`

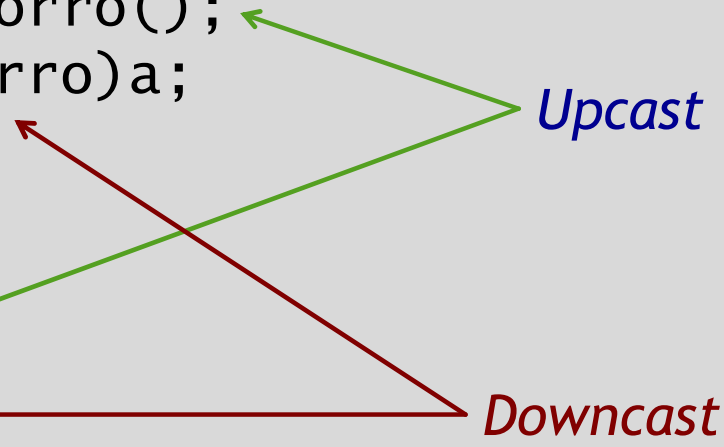


Upcast vs. *downcast*

- Ampliação é automática e livre de erros:
  - A classe base não pode possuir uma interface maior do que a classe derivada;
  - Não é necessário explicitar o upcast.
- Estreitamento é manual e pode causar erros:
  - A classe base pode ter várias subclasses e você está convertendo para a classe errada;
  - É necessário explicitar o downcast;
  - Pode lançar um erro (`ClassCastException`);
  - Pode haver perda de informação (tipos primitivos).

Upcast vs. *downcast*

```
public class Teste {
  public static void main(String[] args) {
    Animal a = new Cachorro();
    Cachorro c = (Cachorro)a;
    c.latir();

    // Forma resumida:
    a = new Gato();
    ((Gato)a).miar();
  }
}
```



nemo

RTTI: Run-Time Type Identification

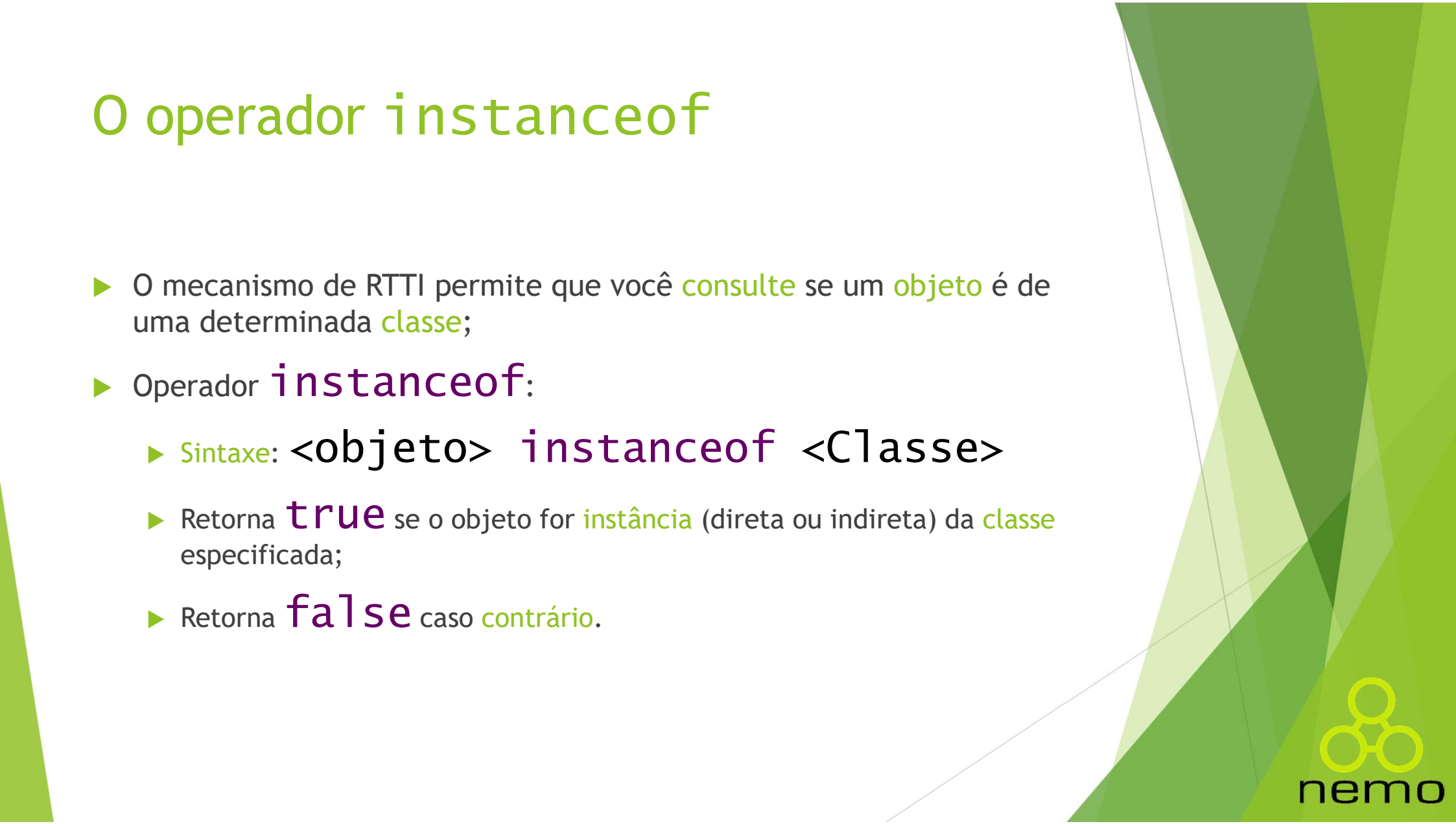
- O mecanismo que verifica o tipo de um objeto em tempo de execução chama-se RTTI;
- RTTI = *Run-Time Type Identification* ou Identificação de Tipos em Tempo de Execução;
- Este mecanismo garante que as conversões são sempre seguras;
- Não permite que um objeto seja convertido para uma classe inválida:
  - Fora da hierarquia: erro de compilação;
  - Dentro da hierarquia: erro de execução.

RTTI: Run-Time Type Identification

```
public class Teste {
  public static void main(String[] args) {
    Animal a = new Cachorro();

    // Sem erro nenhum:
    Cachorro c = (Cachorro)a;

    // Erro de execução (ClassCastException):
    Gato g = (Gato)a;

    // Erro de compilação:
    String s = (String)a;
  }
}
```

O operador `instanceof`

- O mecanismo de RTTI permite que você consulte se um objeto é de uma determinada classe;
- Operador `instanceof`:
  - Sintaxe: `<objeto> instanceof <Classe>`
  - Retorna `true` se o objeto for instância (direta ou indireta) da classe especificada;
  - Retorna `false` caso contrário.

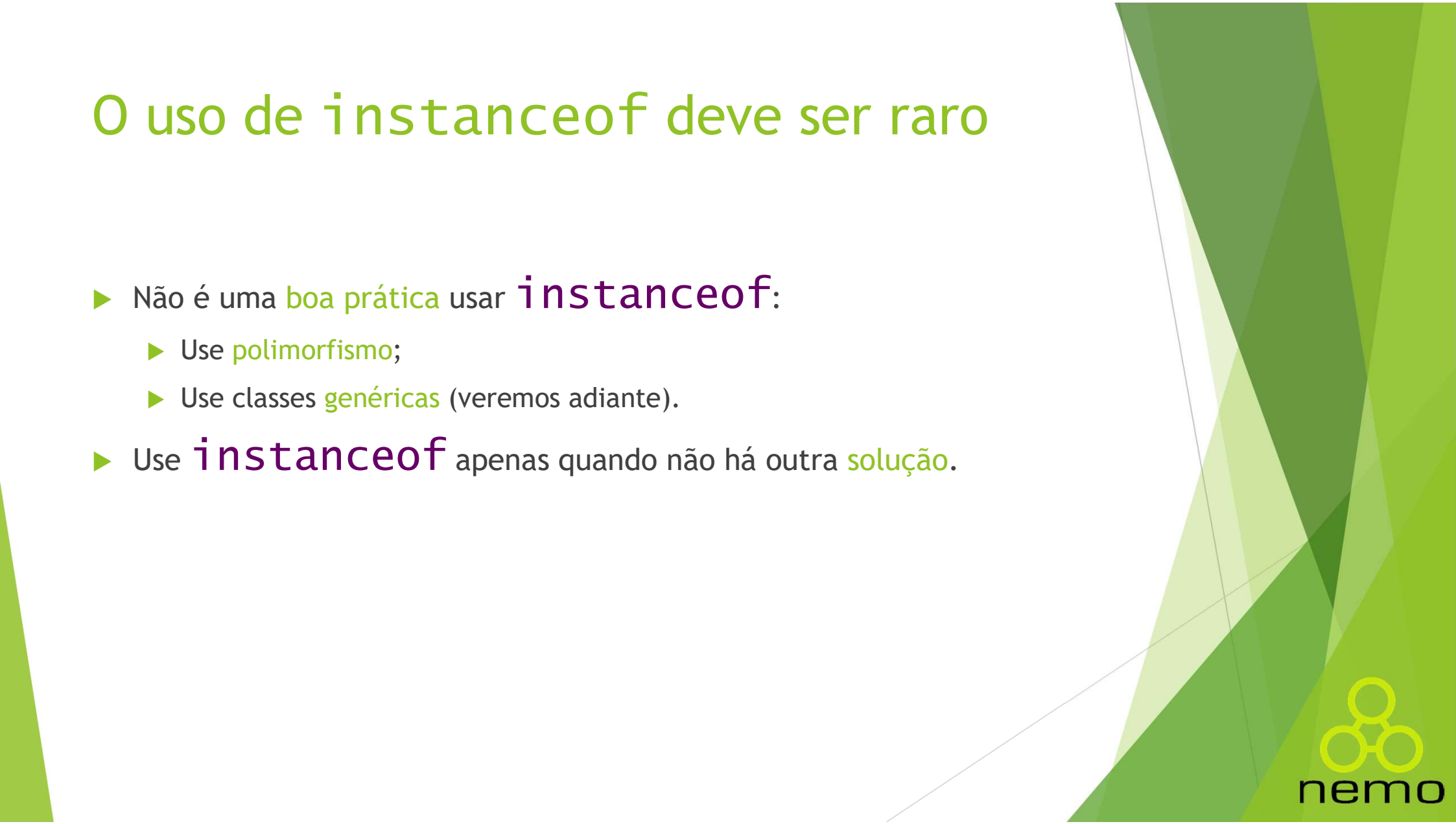


O operador instanceof

```
public class Teste {
  public static void main(String[] args) {
    Animal[] vet = new Animal[] {
        new Cachorro(), new Gato(),
        new Gato(), new Cachorro()
    };

    for (int i = 0; i < vet.length; i++) {
      if (vet[i] instanceof Cachorro)
        ((Cachorro)vet[i]).latir();
      else if (vet[i] instanceof Gato)
        ((Gato)vet[i]).miar();
    }
  }
}
```

O uso de `instanceof` deve ser raro

- Não é uma boa prática usar `instanceof`:
  - Use polimorfismo;
  - Use classes genéricas (veremos adiante).
- Use `instanceof` apenas quando não há outra solução.



Trocando `instanceof` por polimorfismo

```
interface Animal {
  void comer();
  void falar();
}

class Cachorro extends Animal {
  @Override public void comer() { /* ... */ }
  @Override public void falar() { /* ... */ }
}

class Gato extends Animal {
  @Override public void comer() { /* ... */ }
  @Override public void falar() { /* ... */ }
}
```

Trocando instanceof por genéricos

```
public class Teste {
  public static void main(String[] args) {
    Cachorro c;

    List lista = new ArrayList();
    lista.add(new Cachorro());
    Object o = lista.get(0);
    if (o instanceof Cachorro) c = (Cachorro)o;

    // Com genéricos.
    List<Cachorro> listaGen;
    listaGen = new ArrayList<Cachorro>();
    listaGen.add(new Cachorro());
    c = listaGen.get(0);
  }
}
```

Interface Comparable

Interface Comparable

- Ao implementar a interface **comparable**, a classe precisa sobreescrever o método **compareTo**.

```
class Jogo implements Comparable<Jogo> {

  @Override
  public int compareTo(Jogo v) {
     // Implementação
  }
}
```

Interface Comparable

- O programador pode então escolher quais atributos serão comparados, para determinar se um objeto é maior (1), menor (-1) ou igual (0) a outro objeto.

```
class Filme implements Comparable<Filme> {

  @Override
  public int compareTo(Filme f) {
     if(this.anoLancamento > j. getAnoLancamento()) return 1;
     else if(this.anoLancamento < j. getAnoLancamento()) return -1;
     return 0;
  }
}
```

Interface Comparable

- É possível agora realizar comparações entre objetos. Ex:

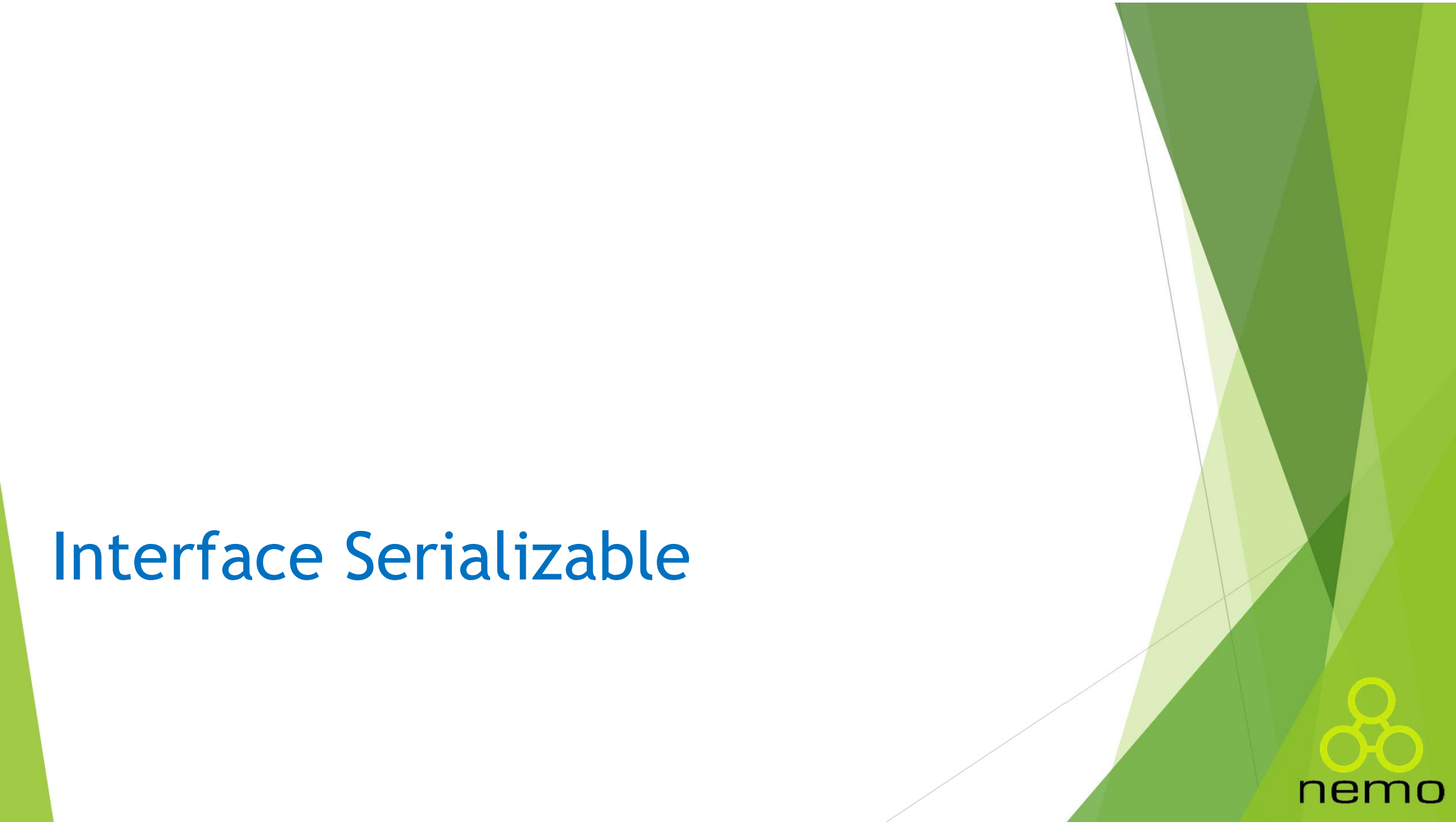
```
public class Main {

  public static void main(String[] args) {

     Filme f1, f2;
     f1 = new Filme (“Thor”,2015);
     f2 = new Filme (“Vingadores”,2019);
     if (f1 > f2) { // false
        System.out.println(“F1 eh maior!”);
     } else { // true
        System.out.println(“F2 eh maior!”);
     }
  }
}
```

Interface Comparable

- Outra grande vantagem de implementar a interface Comparable, é que possível ordenar uma lista desse tipo de objeto utilizando a função Collections.sort();

```
public class Main {

  public static void main(String[] args) {

     Filme f1, f2;
     f1 = new Filme (“Thor”,2015);
     f2 = new Filme (“Vingadores”,2019);
     ArrayList<Filme> filmes = new ArrayList();
     filmes.add(f1);
     filmes.add(f2);
     Collections.sort(filmes); //A lista é ordenada
seguindo o critério definido no método compareTo
}
```

Interface Serializable



Interface Serializable

- Ao implementar essa interface, nenhum método deve ser sobrescrito.
- Qual a função dessa interface então?
- Ela permite que um objeto seja salvo em um arquivo!
- Mas não simplesmente salvar os atributos do objeto separadamente como é feito em C++
- E sim salvar a referencia do objeto no arquivo!

```
class Filme implements Serializable {

}
```

Arquivo de Escrita

- Para **salvar** objetos dentro de um arquivo é necessário definir o **arquivo de saída** e o **stream** que será responsável por salvar os objetos.

```
try {
    FileOutputStream file = new FileOutputStream("Filmes.ser");
    ObjectOutputStream o = new ObjectOutputStream(file);
    for (Filme f : filmes) {
        o.writeObject(f);
    }
    o.close();
    file.close();
} catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

- Vale ressaltar que algumas exceções podem ser geradas durante a manipulação do arquivo.

Arquivo de Leitura

- Para **carregar** objetos dentro de um arquivo é necessário definir o **arquivo de entrada** e o **stream** que será responsável por salvar os objetos.

```
try {
    FileInputStream file = new FileInputStream("Filmes.ser");
    ObjectInputStream o = new ObjectInputStream(file);
    filmes = new ArrayList();
    while(true) {
        filmes.add((Filme)o.readObject());
    }
}catch (EOFException ex1) {
    System.out.println("Filmes carregados com sucesso!");
    break; //EOF reached.
}catch (FileNotFoundException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

- É necessário realizar um cast do objeto lido.
- Quando a exceção EOFException é capturada, isso indica que a leitura foi finalizada.

That’s all Folks!