

**nemo**

ontology & conceptual  
modeling research group



# Desenvolvimento OO com Java Herança, reescrita e polimorfismo

Vítor E. Silva Souza

([viktor.souza@ufes.br](mailto:viktor.souza@ufes.br))

<http://www.inf.ufes.br/~vitorsouza>

Departamento de Informática

Centro Tecnológico

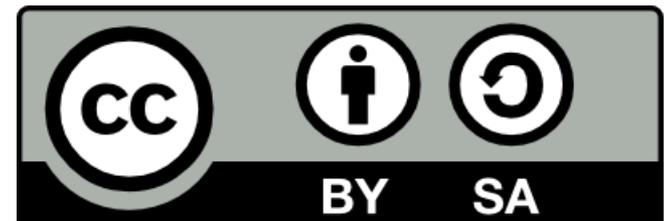
Universidade Federal do Espírito Santo

# Licença para uso e distribuição

- Este obra está licenciada com uma licença Creative Commons Atribuição-Compartilhalgual 4.0 Internacional;
- Você tem o direito de:
  - Compartilhar: copiar e redistribuir o material em qualquer suporte ou formato
  - Adaptar: remixar, transformar, e criar a partir do material para qualquer fim, mesmo que comercial.
- De acordo com os termos seguintes:
  - Atribuição: você deve dar o crédito apropriado, prover um link para a licença e indicar se mudanças foram feitas. Você deve fazê-lo em qualquer circunstância razoável, mas de maneira alguma que sugira ao licenciante a apoiar você ou o seu uso;
  - Compartilhalgual: se você remixar, transformar, ou criar a partir do material, tem de distribuir as suas contribuições sob a mesma licença que o original.



Mais informações podem ser encontradas em:  
<http://creativecommons.org/licenses/by-sa/4.0/>



- O que é Java;
  - Variáveis primitivas e controle de fluxo;
  - Orientação a objetos básica;
  - Um pouco de vetores;
  - Modificadores de acesso e atributos de classe;
- ➔ Herança, reescrita e polimorfismo;
- Classes abstratas;
  - Interfaces;
  - Exceções e controle de erros;
  - Utilitários da API Java.

---

Estes slides foram baseados na [apostila do curso FJ-11: Java e Orientação a Objetos da Caelum](#) e na apostila Programação Orientada a Objetos em Java do [prof. Flávio Miguel Varejão](#).

- Para entregar software de **qualidade** em menos **tempo**, é preciso **reutilizar**;
  - “Copiar & colar” **não** é reuso!
- Reuso é uma das principais **vantagens** anunciadas pela Orientação a **Objetos**;
- Mecanismo **baseado** no conceito de **classes**:
  - **Composição** (“tem um”);
  - **Herança** ou derivação (“é um”).

- Criação de uma **nova** classe usando classes **existentes** como **atributos**;
- Relacionamento “**tem um**”: uma conta tem um titular (cliente), um cliente tem um nome (*string*);
- **Vimos** como fazer isso anteriormente:
  - Atributos **primitivos** e referências a **objetos**;
  - Operadores de **seleção**;
  - **Inicialização** (zerar e atribuir valor inicial);
  - O valor **null**;
  - Atributos **estáticos**.

- 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.

- Classes com **elementos** (atributos, métodos) **repetidos**:

```
class Produto {  
    String nome;  
    double preco;  
  
    Produto() { } // Precisa?  
  
    public Produto(String nome, double preco) {  
        this.nome = nome; this.preco = preco;  
    }  
  
    public boolean ehCaro() {  
        return (preco > 100);  
    }  
  
    // Eventuais outros métodos...  
}
```

- Classes com **elementos** (atributos, métodos) **repetidos**:

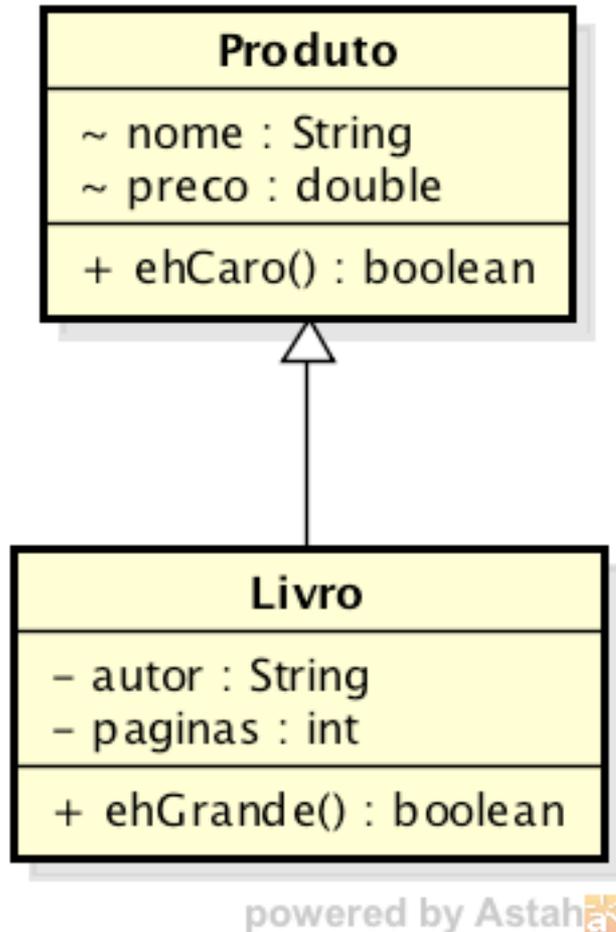
```
class Livro {  
    String nome;  
    double preco;  
    String autor;  
    int paginas;  
  
    public Livro(String n, double p, String a, int pg) {  
        nome = n; preco = p; autor = a; paginas = pg;  
    }  
  
    public boolean ehCaro() { return (preco > 100); }  
  
    public boolean ehGrande() { return (paginas > 200); }  
  
    // Eventuais outros métodos...  
}
```

Não escreva assim, é só pra caber no slide!

- Código **repetido** = problema de **manutenção**;
  - Se surge um novo tipo de produto?
  - Se muda alguma coisa em todos os produtos?
- Colocar os atributos **extras** em Produto, porém só **utilizá-los** em objetos que representem **livros**?
  - Solução **confusa**, desperdiça **memória**, ainda mais se a hierarquia crescer (CDs, DVDs, eletrônicos, etc.);
- Usar **composição**?
  - Também causa **confusão**. Um livro **tem** um produto ou um livro **é** um produto?
- Solução OO: **herança**!

- Livro **estende** produto (adiciona novos membros):

```
class Livro extends Produto {  
    //private String nome;           // Não preciso repetir.  
    //private double preco;         // Herdo de Produto  
    private String autor;  
    private int paginas;  
  
    public Livro(String n, double p, String a, int pg) {  
        nome = n; preco = p; autor = a; paginas = pg;  
    }  
  
    // Também não preciso repetir:  
    // public boolean ehCaro() { return (preco > 100); }  
  
    public boolean ehGrande() { return (paginas > 200); }  
  
    // Eventuais outros métodos...  
}
```



Um livro é um  
(tipo de) produto.

- Podemos chamar métodos do Produto no Livro:

```
public class Loja {  
    public static void main(String[] args) {  
        Livro l = new Livro("Linguagens de Programação",  
            74.90, "Flávio Varejão", 334);  
  
        System.out.println(l.ehCaro());  
        System.out.println(l.ehGrande());  
    }  
}
```

Produto

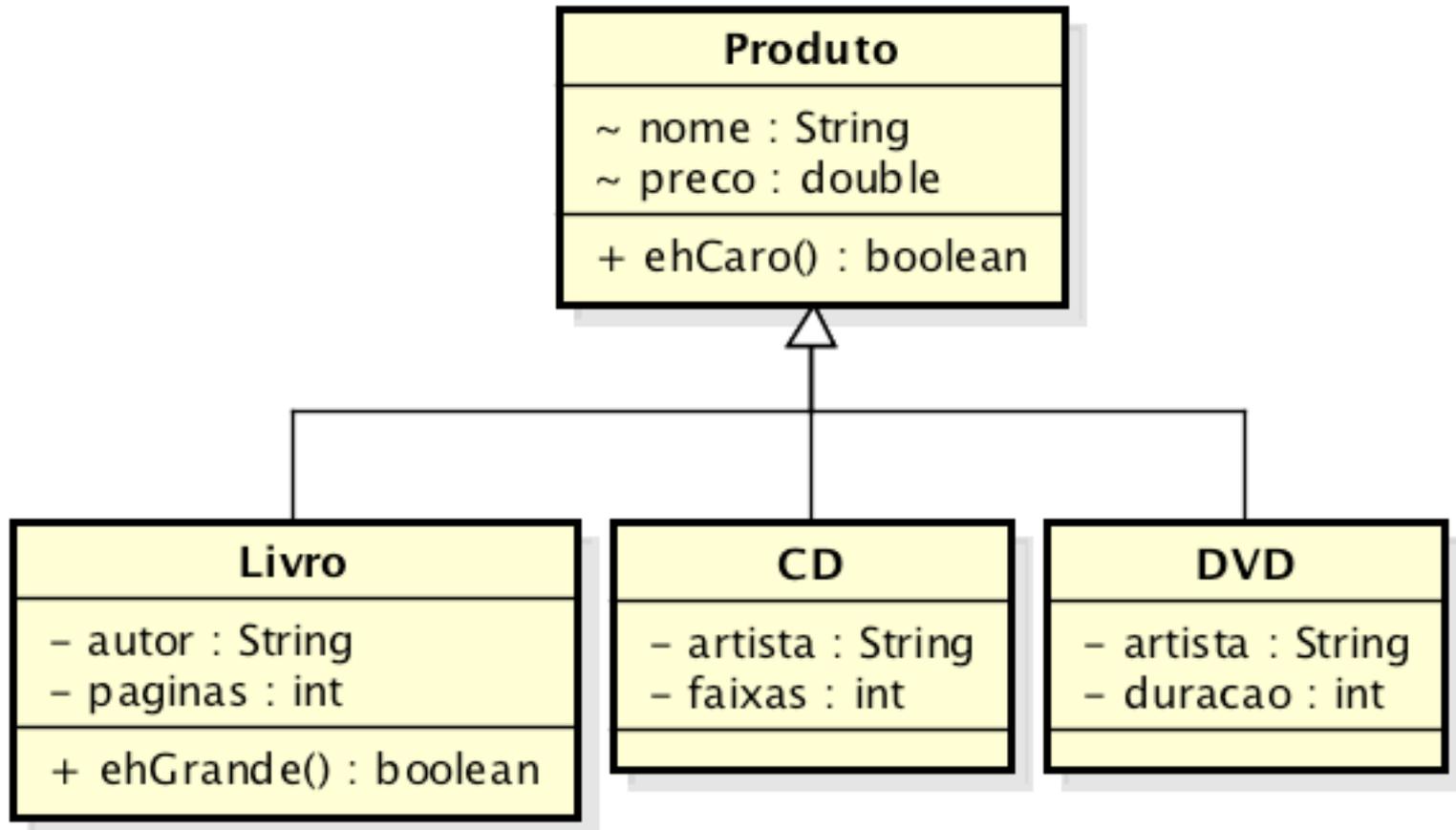
- Superclasse;
- Classe base;
- Classe pai/mãe;
- Classe ancestral, etc.

Livro

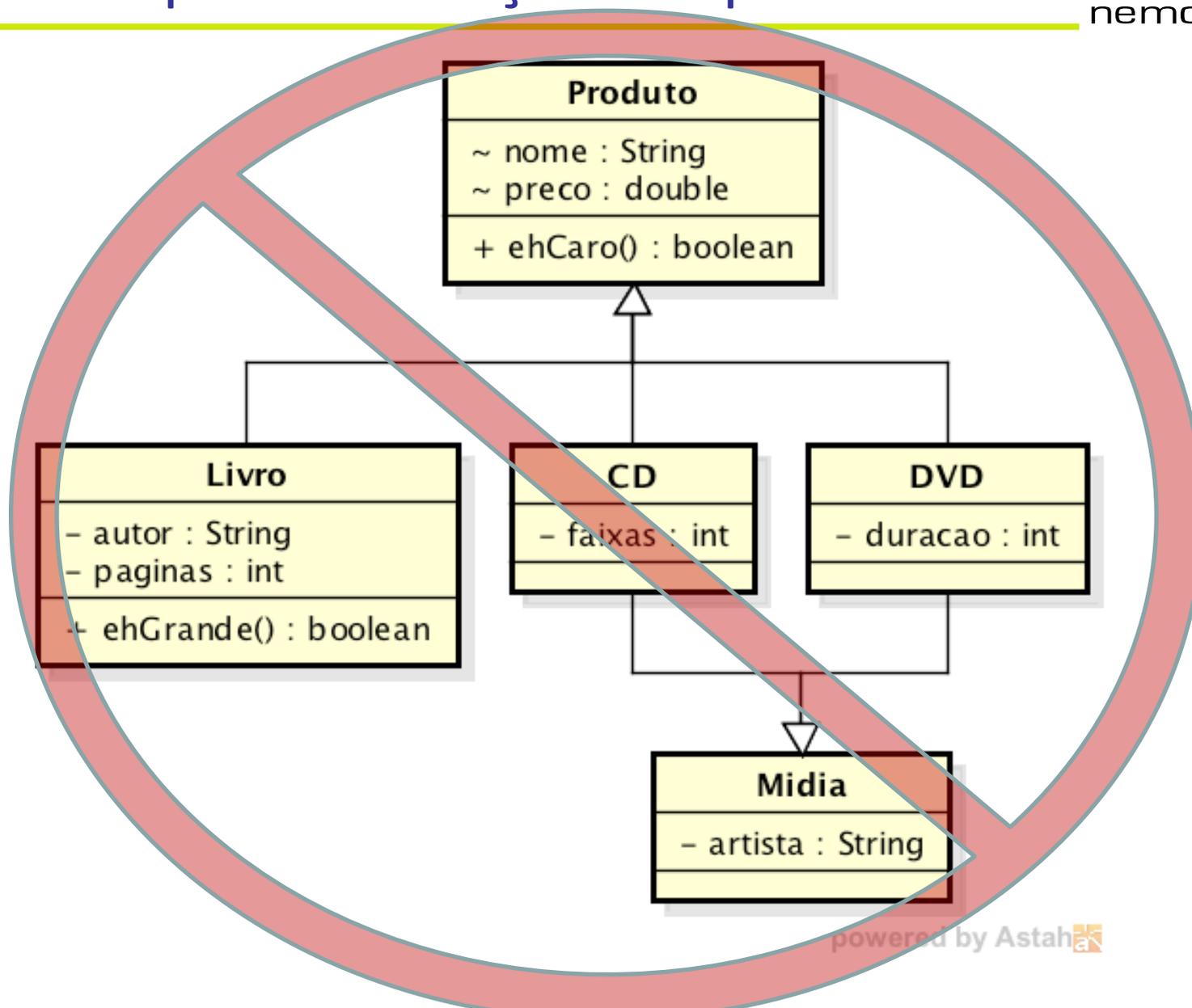
- Subclasse;
- Classe derivada;
- Classe filha;
- Classe descendente, etc.

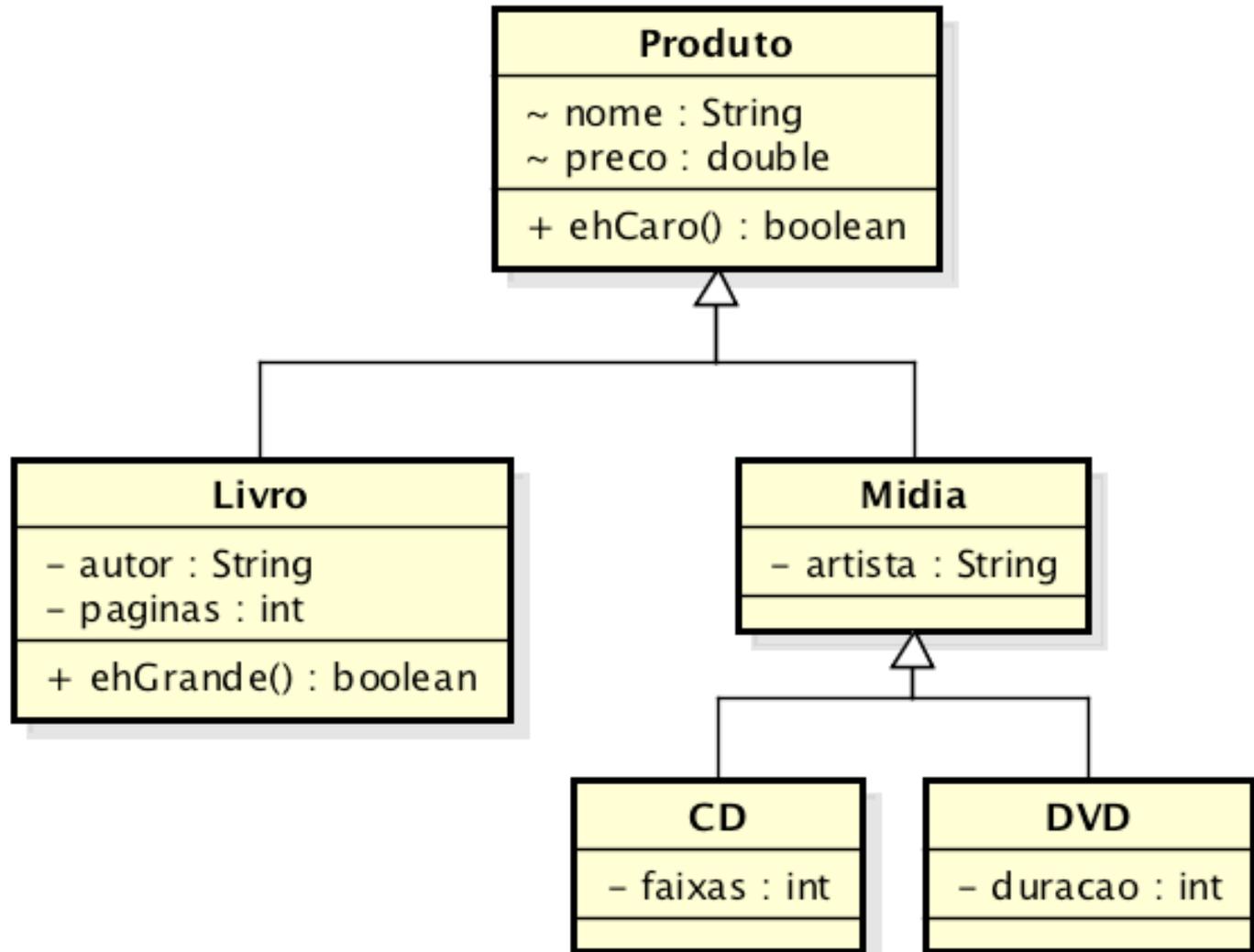
# Java suporta herança simples

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



# Java não suporta herança múltipla





- Sintaxe:

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

- Semântica:

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

- Livro possui autor e paginas (definidos na **própria** classe);
- Livro possui nome e preco (definidos na **superclasse**);
- Livro pode receber mensagens ehGrande() (definida na **própria** classe);
- Livro pode receber mensagens ehCaro() (definida na **superclasse**).

E se nome e preco fossem definidos como privados?

# Visibilidade dos atributos herdados

- Acesso **privativo** é só para a **própria classe**:

```
class Produto {  
    private String nome;  
    private double preco;  
  
    // Restante da classe...  
}
```

```
class Livro extends Produto {  
    private String autor;  
    private int paginas;  
  
    public Livro(String n, double p, String a, int pg) {  
        nome = n; preco = p; autor = a; paginas = pg;  
    }  
}
```

```
// Res
```

```
error: nome has private access in Produto  
    nome = n; preco = p; autor = a; paginas = pg;  
    ^
```

- Utilizamos o acesso **protegido**:

```
class Produto {  
    protected String nome;  
    protected double preco;  
  
    // Restante da classe...  
}
```

```
class Livro extends Produto {  
    private String autor;  
    private int paginas;
```

```
    public Livro(String n, double p, String a, int pg) {  
        nome = n; preco = p; autor = a; paginas = pg;  
    }
```

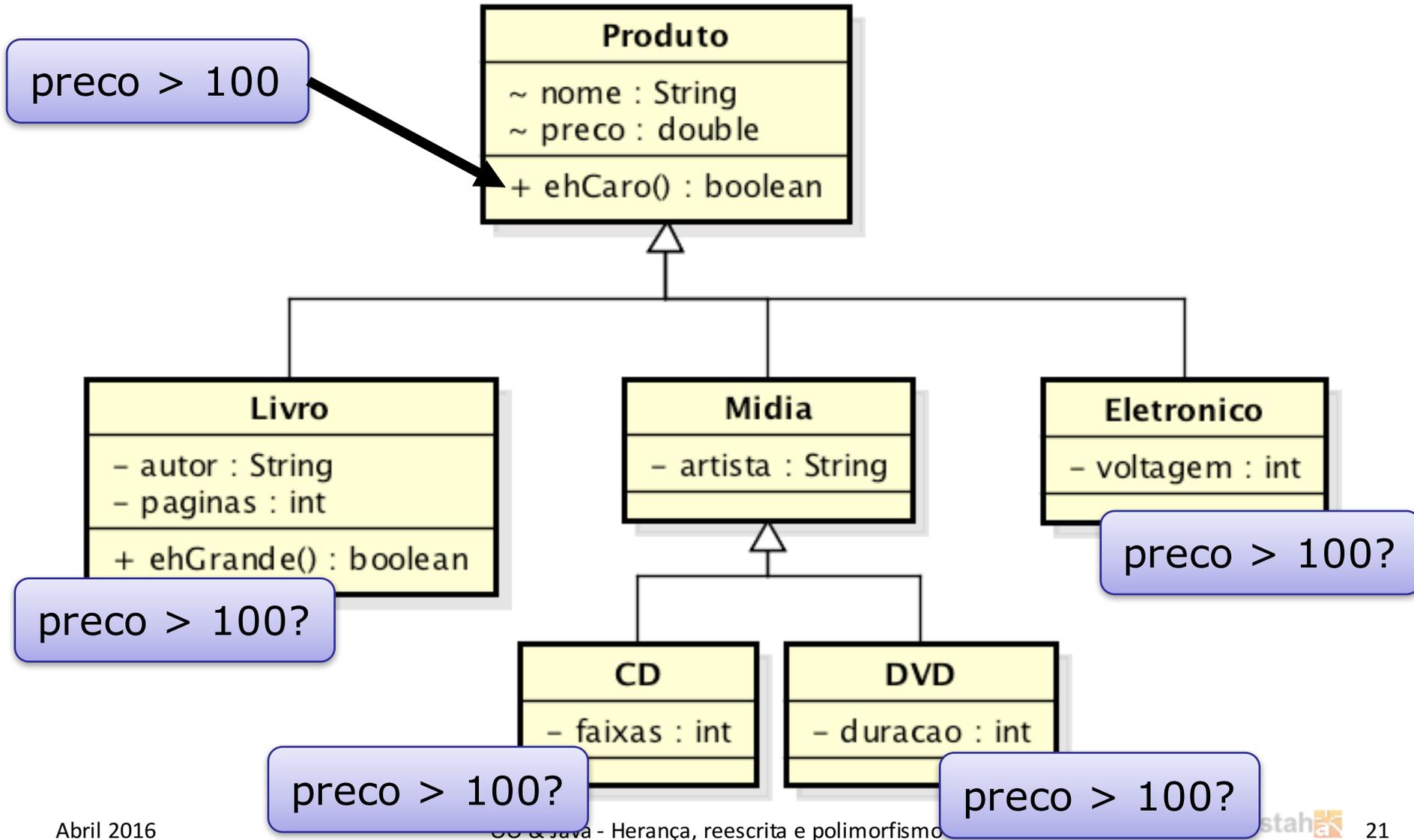
```
    // Restante da classe...  
}
```

Atenção! O acesso protegido não deveria ser usado automaticamente junto com a herança. Faz sentido a filha acessar atributos da mãe?

Acesso	Público	Protegido	Amigo	Privado
A própria classe	Sim	Sim	Sim	Sim
Classe no mesmo pacote	Sim	Sim	Sim	Não
Subclasse em pacote diferente	Sim	Sim	Não	Não
Não-subclasse em pacote diferente	Sim	Não	Não	Não

# Reescrita/sobrescrita de método

- Um método herdado pode não fazer total sentido:



- Um método herdado pode não fazer total sentido:

```
public class Loja {  
    public static void main(String[] args) {  
        Eletronico tv = new Eletronico("TV 40 pol.", 200.0);  
  
        // TV 40 pol. por R$ 200? Uma pechincha!  
        System.out.println(tv.ehCaro()); // true (??)  
    }  
}
```

- Se um método herdado não satisfaz, podemos redefini-lo (reescrevê-lo / sobrescrevê-lo):

```
class Eletronico extends Produto {  
    /* Definições anteriores... */  
  
    // Eletronicos acima de R$ 1.000,00 são caros!  
    @Override  
    public boolean ehCaro() {  
        return (preco > 1000);  
    }  
}
```

Que @&#%\$ é essa de @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**.

Quanto mais cedo detectamos erros, melhor!

- Cuidado para não confundir:

```
class Produto {  
    /* ... */  
    public void darDesconto(double valor) {  
        return (preco > 100);  
    }  
}  
  
class Livro extends Produto {  
    public void darDesconto(int valor) { // Foi feita  
        return (preco > 100);          // sobrecarga,  
                                        // não sobrescrita!  
    }  
}
```



@Override aqui  
cairia bem...

- Cuidado para não confundir:

```
class Produto {
    /* ... */
    public void darDesconto(double valor) {
        return (preco > 100);
    }
}

class Livro extends Produto {
    @Override
    public void darDesconto(int valor) { // Foi feita
        return (preco > 100);           // sobrecarga,
                                        // não sobrescrita!
    }
}
```

error: method does not override or  
implement a method from a supertype  
@Override  
^

- 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”.
- Fizemos isso quando **definimos** mais de um **construtor** para nossa classe!
- Podemos **usar** este conceito para **qualquer** método.

- Como Java **distingue** entre dois métodos com o mesmo nome?
  - Pelos **tipos** dos **parâmetros**!
  - Já vimos que 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) { /* ... */ }
```

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

# Sobrecarga de valor de retorno?

- O valor de **retorno** de um método não é incluído em sua assinatura. Por que?

```
public class SobreRetorno {  
    static int retorna10() { return 10; }  
    static double retorna10() { return 10.0; }  
  
    public static void main(String[] args) {  
        int x = retorna10();    // OK!  
        double d = retorna10();    // OK!  
  
        // Qual método chamar?  
        System.out.println(retorna10());  
    }  
}
```

- 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

- Posso fazer o mesmo com construtores:

```
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()...

- Se um Livro é um Produto, para criarmos um livro precisamos antes criar um produto.

```
class Produto {
    String nome;
    double preco;

    Produto() {
        System.out.println("Criando produto sem nome");
    } // Precisa?

    public Produto(String nome, double preco) {
        System.out.println("Criando produto: " + nome);
        this.nome = nome; this.preco = preco;
    }

    // Restante da classe...
}
```

- Se um Livro é um Produto, para criarmos um livro precisamos antes criar um produto.

```
class Livro extends Produto {  
    private String autor;  
    private int paginas;  
  
    public Livro(String n, double p, String a, int pg) {  
        System.out.println("Criando livro: " + n);  
        nome = n; preco = p; autor = a; paginas = pg;  
    }  
  
    // Restante da classe...  
}
```

- Se um Livro é um Produto, para criarmos um livro precisamos antes criar um produto.

```
public class Loja {  
    public static void main(String[] args) {  
        Livro l = new Livro("Linguagens de Programação",  
                             74.90, "Flávio Varejão", 334);  
  
        // O que vai imprimir?  
    }  
}
```

Criando produto sem nome  
Criando livro: Linguagens de Programação

- O construtor sem argumentos da classe base é chamado **implicitamente**:

```
Livro l = new Livro("Linguagens de Programação",  
    74.90, "Flávio Varejão", 334);
```

```
public Livro(String n, double p, String a, int pg) {  
    // Chamada implícita: super();  
    System.out.println("Criando livro: " + n);  
    nome = n; preco = p; autor = a; paginas = pg;  
}
```

```
Produto() {  
    System.out.println("Criando produto sem nome");  
} // Precisa!!!
```

- E se o construtor sem argumentos não existir?

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

class Livro extends Produto { /* ... */
    public Livro(String n, double p, String a, int pg) {
        nome = n; preco = p; autor = a; paginas = pg;
    }
}
```

error: constructor Produto in class Produto cannot be applied to given types;

```
    public Livro(String n, double p, String a, int pg) {
                                                    ^
```

required: String,double

found: no arguments

reason: actual and formal argument lists differ in length

```
class Pessoa {  
    private String nome;  
    public Pessoa(String nome) {  
        this.nome = nome;  
    }  
}
```

```
class Aluno extends Pessoa { }
```

```
// cannot find symbol  
// symbol : constructor Pessoa()  
// location: class Pessoa  
// class Aluno extends Pessoa {  
// ^  
// 1 error
```

- Aluno não define construtor: **ganha** um *default*;
- Pessoa define um construtor com parâmetro: **não ganha** construtor default;
- Construtor default **tenta** chamar construtor sem parâmetro na superclasse (Pessoa);
- Pessoa **não possui** construtor sem parâmetro!

- A **solução** é chamar o construtor **explicitamente**;
- Assim como **this()**, **super()** deve ser o **primeiro** comando do construtor.

```
class Aluno extends Pessoa {  
    public Aluno() {  
        super("Sem nome");  
    }  
  
    public Aluno(String nome) {  
        super(nome);  
    }  
}
```

- Use **herança** quando:
  - Uma classe representa um **subtipo** de outra classe;
  - Construção de **famílias** de tipos;
  - Use com **cuidado!**
- Use **composição** quando:
  - Uma classe representa algo que **faz parte** de outra;
  - **Prefira** composição à herança.
- Os dois **conceitos** são utilizados em **conjunto** a todo momento!

```
class Lista {  
    public void adic(int pos, Object obj) { }  
    public Object obter(int pos) { }  
    public void remover(int pos) { }  
}
```

// Uma pilha é uma lista?

```
class Pilha1 extends Lista { }
```

// Ou uma pilha tem uma lista?

```
class Pilha2 {  
    private Lista elementos;  
    public void empilha(Object obj) { }  
    public Object desempilha() { }  
}
```

- De maneira **geral**:
  - **Atributos** de uma classe devem ser **privados**;
  - Se a classe possui filhas, **atributos** podem ser **protegidos** ou possuir **métodos** de acesso protegidos;
  - Métodos que pertencem à **interface** devem ser **públicos**;
  - Alguns **métodos** podem ser utilizados **internamente** e, portanto, serem **privados** ou **protegidos**.

- Suportar do desenvolvimento **incremental**;
  - Classes já **testadas** podem ser **reutilizadas**;
  - Economia de **tempo**.
- Relacionamento “**é um**”:
  - Permite **substituir** a classe base por uma **subclasse** quando a primeira é esperada;
  - Propriedade que chamamos de **polimorfismo**.

Desenvolvimento OO com Java - Herança, reescrita e polimorfismo

# A PALAVRA-CHAVE 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.

- 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;
```

- 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!  
    }  
}
```

```
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();
    }
}
```

- 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 **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 **privados** não podem ser **acessados**;
- Portanto, são  **finais** por natureza (as subclasses não têm acesso a ele).

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

- 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 TelefoneAtomico extends TelefoneCelular { }
```

Desenvolvimento OO com Java - Herança, reescrita e polimorfismo

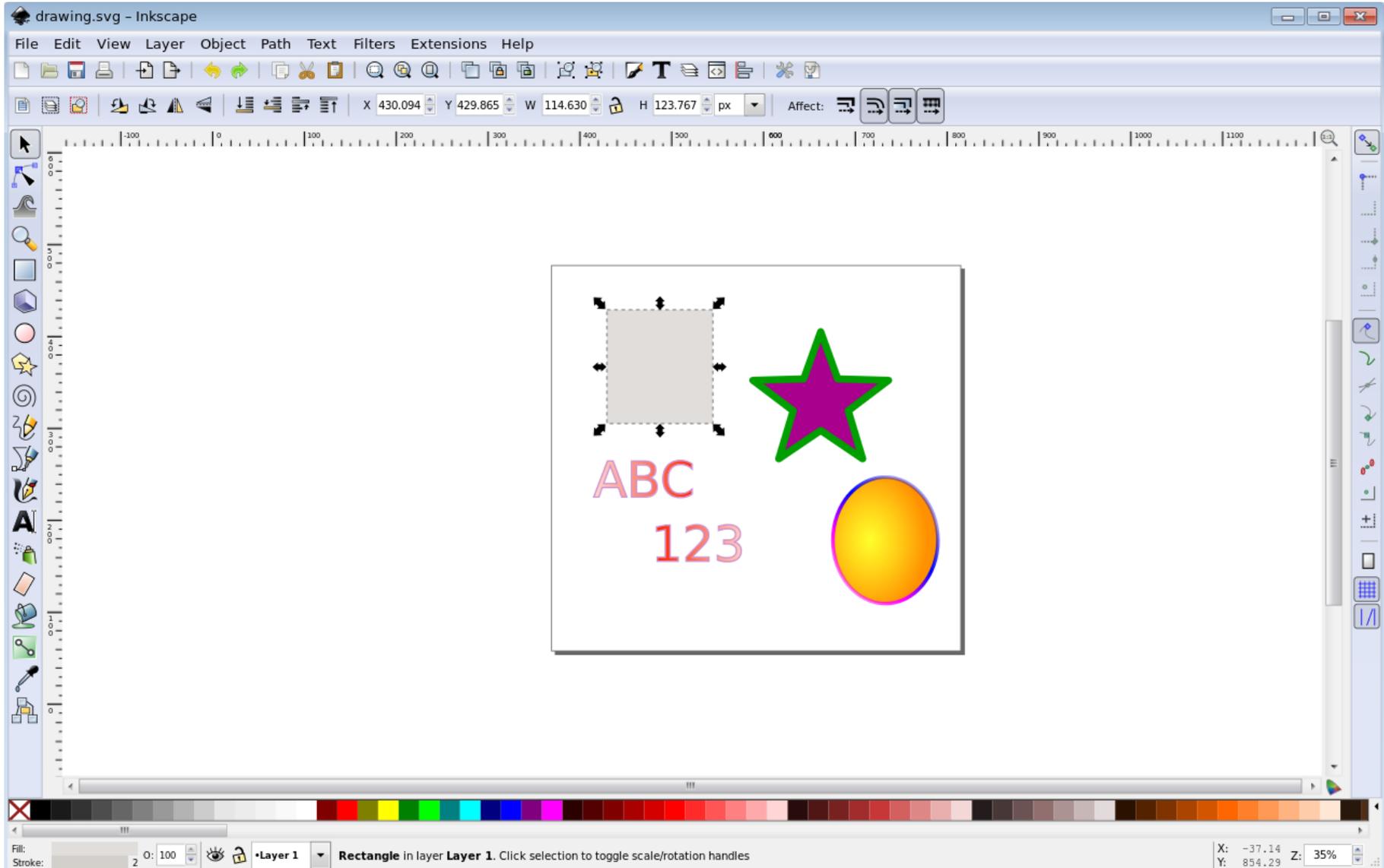
# POLIMORFISMO

- 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**.

“Enviamos nossos produtos para todo o Brasil”

Será que envia DVDs também?

# Exemplo: um aplicativo de desenho



# Exemplo: um aplicativo de desenho

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

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

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

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

# Exemplo: um aplicativo de desenho

- 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*) 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!  
    }  
}
```

- O compilador realmente **não sabe** qual é o **tipo**. Veja um exemplo com geração aleatória:

```
public class AplicativoDesenhoAleatorio {
    public static void main(String[] args) {
        Forma f = null;
        switch((int)(Math.random() * 3)) {
            case 0: f = new Circulo(); break;
            case 1: f = new Quadrado(); break;
            case 2: f = new Triangulo(); break;
            default: f = new Forma();
        }
        f.desenhar();
    }
}
```

- Quando realizamos ampliação, “esquecemos” o tipo de um objeto:

Forma `f = new Quadrado();`

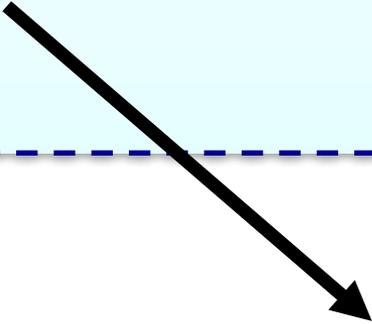
- Não sabemos mais qual é a **classe específica** de `f`. Sabemos **apenas** que ele é uma forma;
- **Por que** fazer isso?

- 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();  
    }  
}
```

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

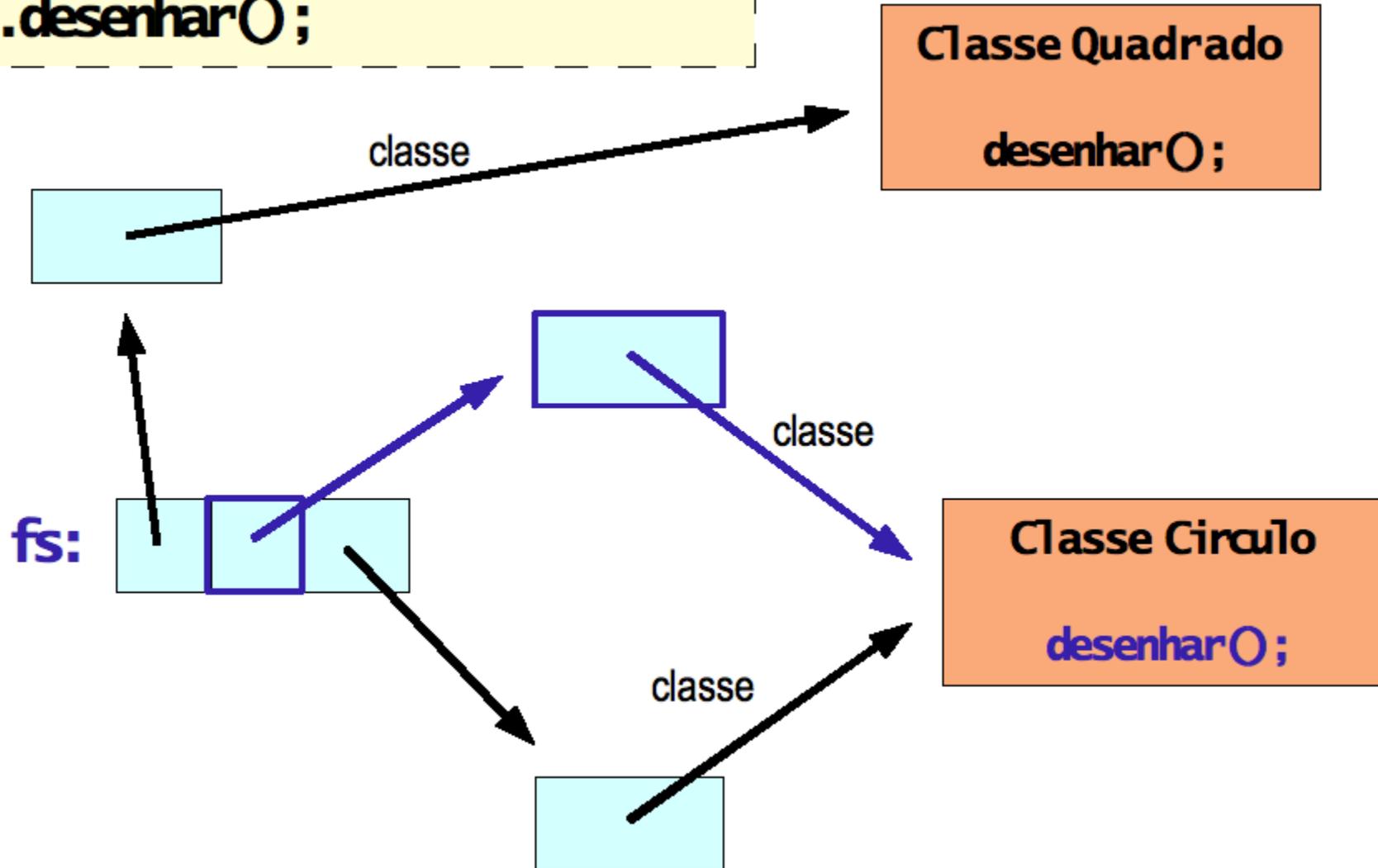
```
public class AplicativoDesenho {  
    private static void desenhar(Forma[] fs) {  
        for (int i = 0; i < fs.length; i++)  
            fs[i].desenhar();  
    }  
}
```



fs[i] é do tipo Forma.  
Chamar sempre Forma.desenhar()?

- 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*.

```
fs[1].desenharO;
```



- 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.

- Extensibilidade:
  - Podemos adicionar **novas** classes sem **alterar** o método polimórfico.

```
class Retangulo extends Forma {  
    public void desenhar() {  
        System.out.println("Retangulo");  
    }  
}  
  
class Quadrado extends Retangulo {  
    public void desenhar() {  
        System.out.println("Quadrado");  
    }  
}
```

```
class Reta extends Forma {
    public void desenhar() {
        System.out.println("Reta");
    }
}

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

    public static void main(String[] args) {
        Forma f = new Reta();
        desenhar(f);
    }
}
```

- A **interface** de todos é definida pela classe **base**;
- Novas classes possuem a **mesma interface**, portanto o sistema **já sabe** lidar com elas;
- Mesmo que todas as classes já existam de princípio, poupa-se **tempo** e **esforço**, codificando um método **único** para todas.

Desenvolvimento OO com Java - Herança, reescrita e polimorfismo

# 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 Produto { }  
  
/* É equivalente a: */  
  
class Produto extends 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;
  - `hashCode()`: função **hash**;
  - `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 {
    /* ... */
    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)
```

- Retorna uma **representação** em String do **objeto** em questão;
- Permite **polimorfismo** em grande escala:
  - Se quisermos **imprimir** um objeto de **qualquer** classe, ele será chamado;
  - Se quisermos **concatenar** um objeto de **qualquer** classe com uma String, ele será chamado.

# 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()

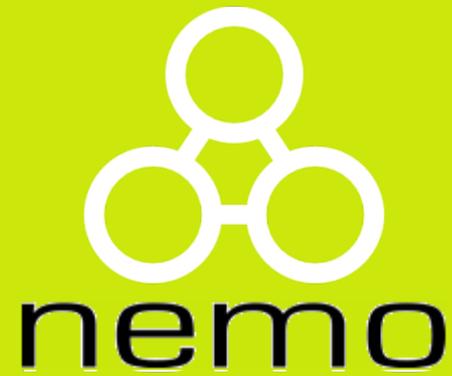
```
class Valor {  
    int i;  
    public Valor(int i) { this.i = i; }  
  
    public boolean equals(Object o) {  
        return (o instanceof Valor)  
            && (((Valor)o).i == i);  
    }  
}
```



```
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  
    }  
}
```

- Compara dois objetos. Retorna **true** se forem iguais/equivalentes, **false** se não forem;
- Permitem **polimorfismo** em grande escala:
  - Podemos criar uma classe **conjunto** que armazena objetos de **qualquer** classe, desde que sejam objetos **diferentes**;
  - Podemos implementar um **método** que permite dizer se um objeto está no **conjunto**, se um conjunto está **contido** em outro, etc.

- Apostila FJ-11 da Caelum:
  - Seção 7.6, página 94 (discussão sobre herança);
  - Seção 7.7, página 95 (conta corrente).



<http://nemo.inf.ufes.br/>