Programação Orientada a Objetos em Java
- Revisão -

Prof. Filipe Mutz

Agenda

- Programação Orientada a Objetos
- Entrada – Processamento - Saída em Java
- Classes e Controle de Acesso
- Herança, Classes Abstratas e Interfaces
- Polimorfismo
- Exceções

Programação Orientada a Objetos

- Objetos representam "coisas" específicas que possuem atributos (características/dados) e são capazes de realizar ações (métodos).
  - O carro de aplicativo que possui placa PBB-9543, pertence à José Almeida da Silva e do qual é possível ler a localização e chamar para corridas.
  - A pessoa chamada "Cláudia Peres Hopfield" com CPF 123123-123 e que é capaz de realizar matrículas e atualizar seu endereço.
  - O livro com título "Cálculo 1", ISBN 123123-123 e código identificador 123 que pode ser emprestado e que tem uma data de devolução (se estiver emprestado).
  - O livro com título "Cálculo 1", ISBN 123123-123 e código identificador 456 que pode ser emprestado e que tem uma data de devolução (se estiver emprestado).

Programação Orientada a Objetos

- Classes definem os atributos e métodos que objetos do mesmo tipo (e.g., os livros do slide anterior) compartilham.
- Sempre que dissermos que um objeto é de um tipo, ele necessariamente terá os atributos e métodos da classe.
- Da mesma forma que é possível construir várias cópias de uma casa a partir de suas plantas, podemos criar objetos a partir da classe. A classe está para as plantas assim como os objetos estão para as casas construídas.

Programação Orientada a Objetos

```
package src;

// Em Java, até os programas são classes
public class Programa
{
	public static void main(String args[]) {

		// Escreva seu programa aqui...
	}
}
```

- Em Java, o programa é definido por uma classe especial que contém o método main.
- O método main define o que o programa irá fazer. Estas ações tipicamente serão orquestradas criando objetos e executando seus métodos.
- Em Java:
- Programas são organizados em projetos.
- Um projeto contém um ou mais pacotes. Cada pacote é uma pasta que pode conter uma ou mais classes.
    - Pacotes servem para organizar o projeto e reduzir a chance de existirem duas classes com o mesmo nome.
- Cada classe é definida em um arquivo cujo nome é igual ao nome da classe. Nomes de classes começam com letra maiúscula.

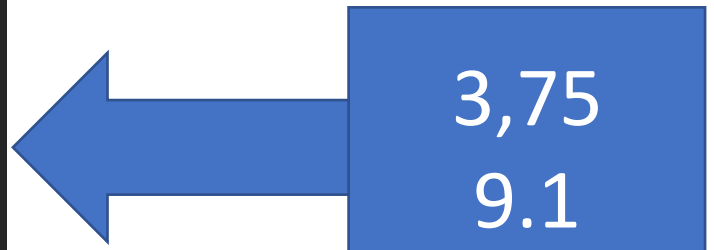
```java
import java.util.ArrayList;

public class Livro {
	String titulo;
	String genero;
	int ano;
	double preco;
	ArrayList<String> autores = new ArrayList<>();
	boolean emprestado = false;

	void devolver() {
		emprestado = false;
	}

	void emprestar() {
		emprestado = true;
	}
}
```

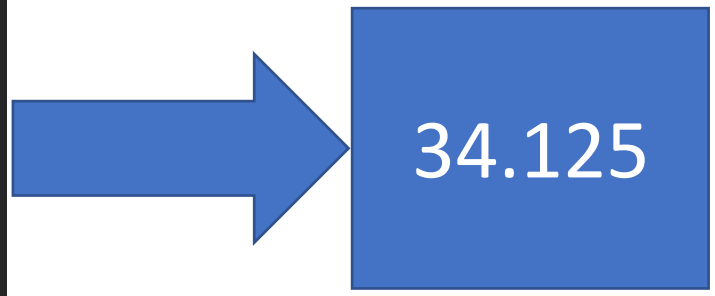
```java
package src;

public class Programa
{
	public static void main(String args[]) {

		Livro l1 = new Livro();
		Livro l2 = new Livro();

		l1.titulo = "Harry Potter";
		l1.genero = "Infanto-juvenil";
		l1.ano = 2000;
		l1.preco = 65.70;
		l1.autores.add("J. K. Howling");

		l1.emprestar();
		l1.devolver();
	}
}
```

Entrada – Processamento - Saída em Java

```
import java.util.Scanner;

public class Programa {
	public static void main(String[] args) {
		Scanner s = new Scanner(System.in);

		System.out.println("Digite o valor da massa: ");
		double massa = s.nextDouble();

		System.out.println("Digite o valor da aceleracao: ");
		double aceleracao = s.nextDouble();

		double forca = massa * aceleracao;

		System.out.println("O valor da forca eh: " + forca);
	}
}
```

```java
public class Programa {
	public static void main(String[] args) {
		try {
			Scanner s = new Scanner(new File("dados.txt"));
			double massa = s.nextDouble();
			double aceleracao = s.nextDouble();
			double forca = massa * aceleracao;

			FileWriter writer = new FileWriter("saida.txt");
			writer.write("A forca eh " + forca + "\n");
			writer.close();

			s.close();

		} catch (FileNotFoundException e) {
			System.out.println("Arquivo 'dados.txt' nao encontrado.");
		} catch (IOException e) {
			System.out.println("Arquivo 'saida.txt' nao pode ser salvo.");
		}
	}
}
```





Exemplo: Uso de Métodos de Strings para Extração de Dados de CSVs

Considere que os dados à direita estão salvos no arquivo “dados.csv”. Faça um programa que conte quantas vezes cada diagnóstico foi dado. Ignore a primeira linha de cabeçalho do arquivo. Para o exemplo ao lado, o programa deve responder normal: 2 e alterado: 3.

```
id;diagnostico
id_1234;normal
id_4567;alterado
id_7892;normal
id_4442;alterado
id_5541;alterado
```

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Scanner;

public class Programa {
    public static void main(String[] args) {
        try {
            Scanner s = new Scanner(new File("dados.csv"));

            // descarta a linha de cabecalho
            s.nextLine();

            // diagns sera' usado para armazenar os diagnosticos diferentes
            // e counts para armazenar quantas vezes o i-esimo diagnostico foi
            // encontrado.
            ArrayList<String> diagns = new ArrayList<>();
            ArrayList<Integer> counts = new ArrayList<>();

(…)
```

```
(…)

            while (s.hasNextLine()) {
                String line = s.nextLine();
                String parts[] = line.split(";");
                int idx = diagns.indexOf(parts[1]);
                if (idx >= 0) {
                    counts.set(idx, counts.get(idx) + 1);
                } else {
                    diagns.add(parts[1]);
                    counts.add(1);
                }
            }

            for (int i = 0; i < diagns.size(); i++)
                System.out.println(diagns.get(i) + ": " + counts.get(i));

        } catch (FileNotFoundException e) {
            System.out.println("Arquivo 'dados.csv' nao encontrado.");
        }
    }
}
```

Após o split, parts[1] irá conter o diagnóstico. O método indexOf buscará o índice de de parts[1] em diagns (-1 se não existir no array). Se o elemento for encontrado, incrementamos a contagem naquela posição. Caso contráro, adicionamos o novo diagnóstico com contagem 1.

```
            s.nextLine();
            ArrayList<String> diagns = new ArrayList<>();
            ArrayList<Integer> counts = new ArrayList<>();

            while (s.hasNextLine()) {
                String line = s.nextLine();
                String parts[] = line.split(";");
                int idx = diagns.indexOf(parts[1]);
                if (idx >= 0) {
                    counts.set(idx, counts.get(idx) + 1);
                } else {
                    diagns.add(parts[1]);
                    counts.add(1);
                }
            }

            for (int i = 0; i < diagns.size(); i++)
                System.out.println(diagns.get(i) + ": " + counts.get(i));

        } catch (FileNotFoundException e) {
            System.out.println("Arquivo 'dados.csv' nao encontrado.");
        }
    }
}
```

```java
public class Programa {
  public static void main(String[] args) {
      Scanner s = new Scanner(System.in);

      System.out.print("Idade: ");
      int idade = s.nextInt();

      while (idade <= 0) {
          System.out.print("Idade invalida. Tente novamente.");
          System.out.print("Idade: ");
          idade = s.nextInt();
      }

      if (idade >= 18)
          System.out.println("Parabens, você já pode tirar a carteira!");
      else
          System.out.println("Aguarde " + (18 - idade) + " anos!");
  }
}
```

```java
import java.util.Scanner;

public class Programa {
    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);

        double soma = 0;
        System.out.print("Digite 5 numeros: ");

        for (int i = 0; i < 5; i++)
            soma += s.nextDouble();

        double media = soma / 5;
        System.out.println("A media eh: " + media);
    }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Controle de Acesso

- Permitir o acesso direto aos atributos:
  - *Exige disciplina dos clientes da classe `Conta`;*
  - *Pode levar a inconsistências;*
- Solução: impedir o acesso externo ao atributo:
  - *Atributo privativo;*
  - *Externo = qualquer outra classe, exceto a proprietária do atributo (ex.: `Conta` para o atributo `saldo`).*
- Vantagens:
  - *Objetos trocam mensagens com base em contratos;*
  - *Modificações na implementação não afetam clientes (ex.: adicionar CPMF nos saques de conta-corrente).*

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

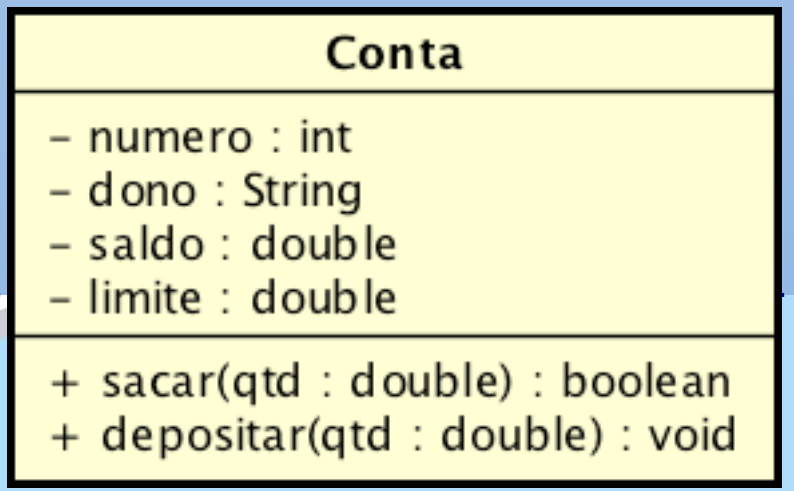
Palavras-chave

- Três palavras-chave especificam o acesso:
  - *public*
  - *private*
  - *protected*
- O nível de acesso *package-private* é determinado pela ausência de especificador;
- Devem ser usadas antes do nome do membro que querem especificar;
- Não podem ser usadas em conjunto.

Implementando o encapsulamento

```
class Conta {
  private int numero;
  private String dono;
  private double saldo;
  private double limite;

  public boolean sacar(double qtd) {
     // ...
  }

  // ...
}
```

Modificador de acesso / visibilidade!



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Sobrecarga (Overloading)

- 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.

```java
public class CalculadoraMulti {

    double multiplicacao(double x, double y) {
        return x + y;
    }

    Matrix multiplicacao(Matrix m, Matrix n) {
        Matrix resultado = new Matrix();

        // multiplicacao de matrizes

        return resultado;
    }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Herança

- 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.*

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Motivação

- 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...
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Motivação

- 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!

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Motivação

- 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 (discos, eletrônicos, cosméticos, etc.);*
- Usar composição?
  - *Também causa confusão. Um livro tem um produto ou um livro é um produto?*
- Solução OO: herança!

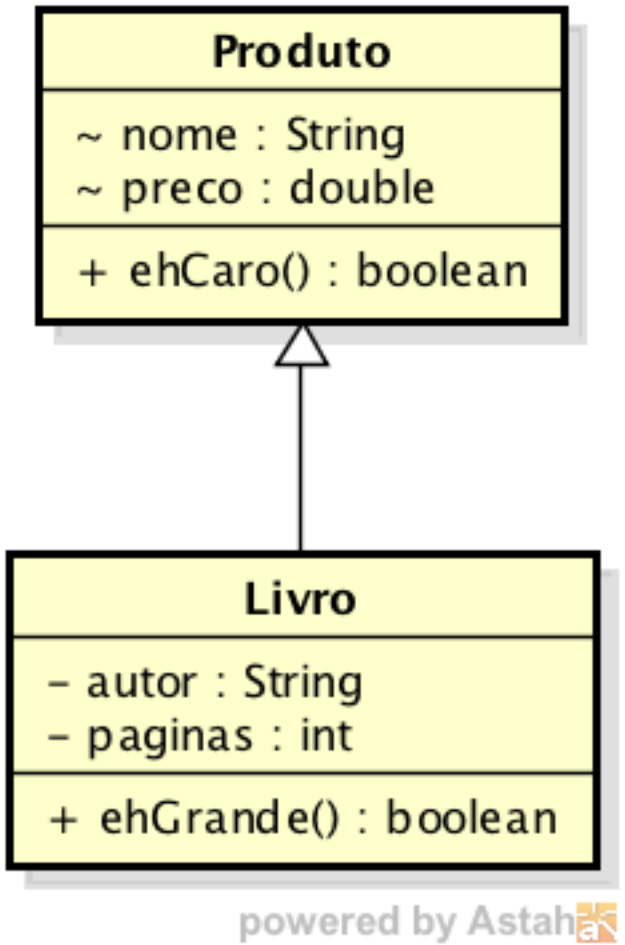
UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Solução com 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...
}
```

Herança em UML



Um livro é um (tipo de) produto.

Solução com herança

- 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:

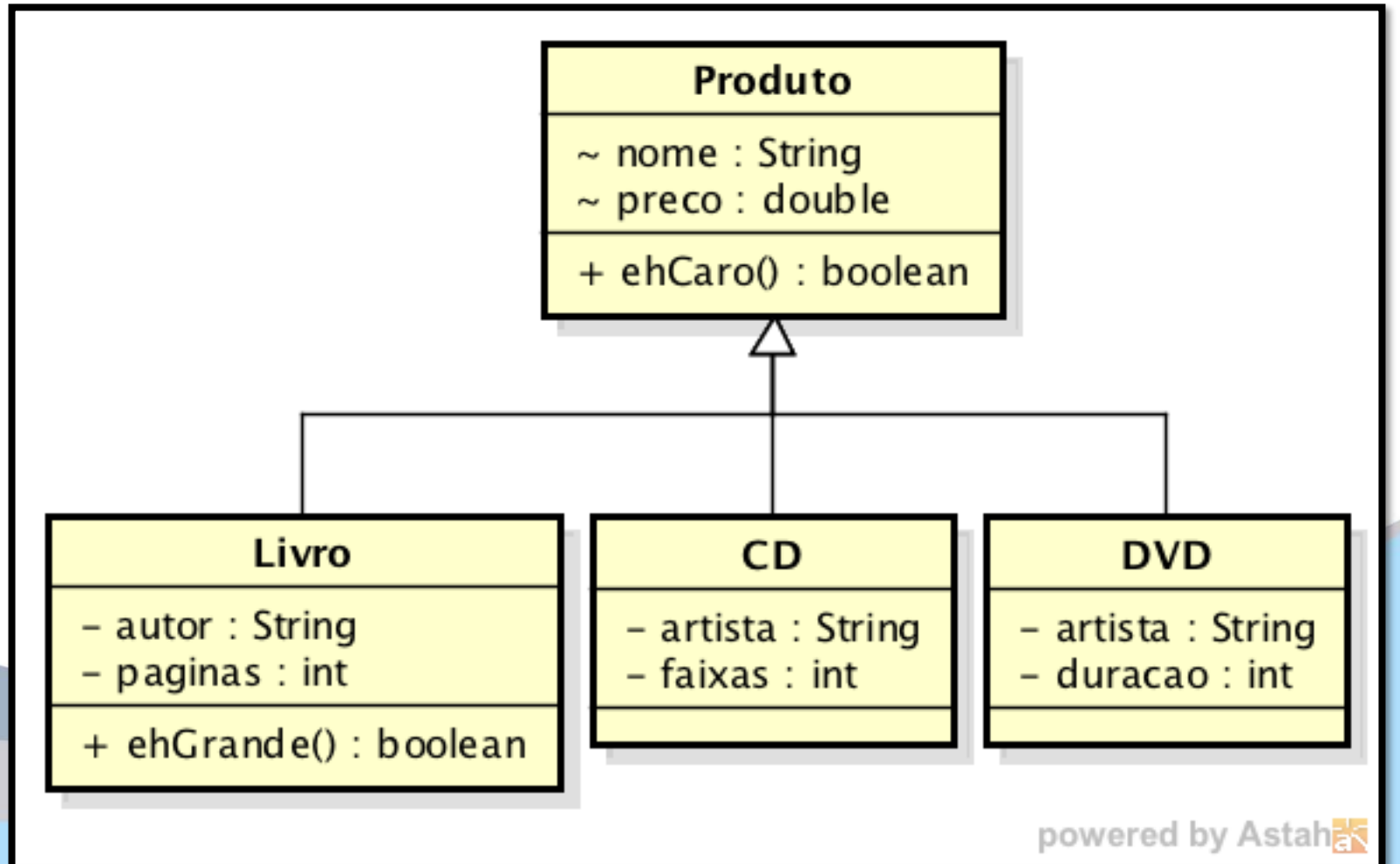
- Superclasse;
- Classe base;
- Classe pai/mãe/ancestral, etc.

Livro:

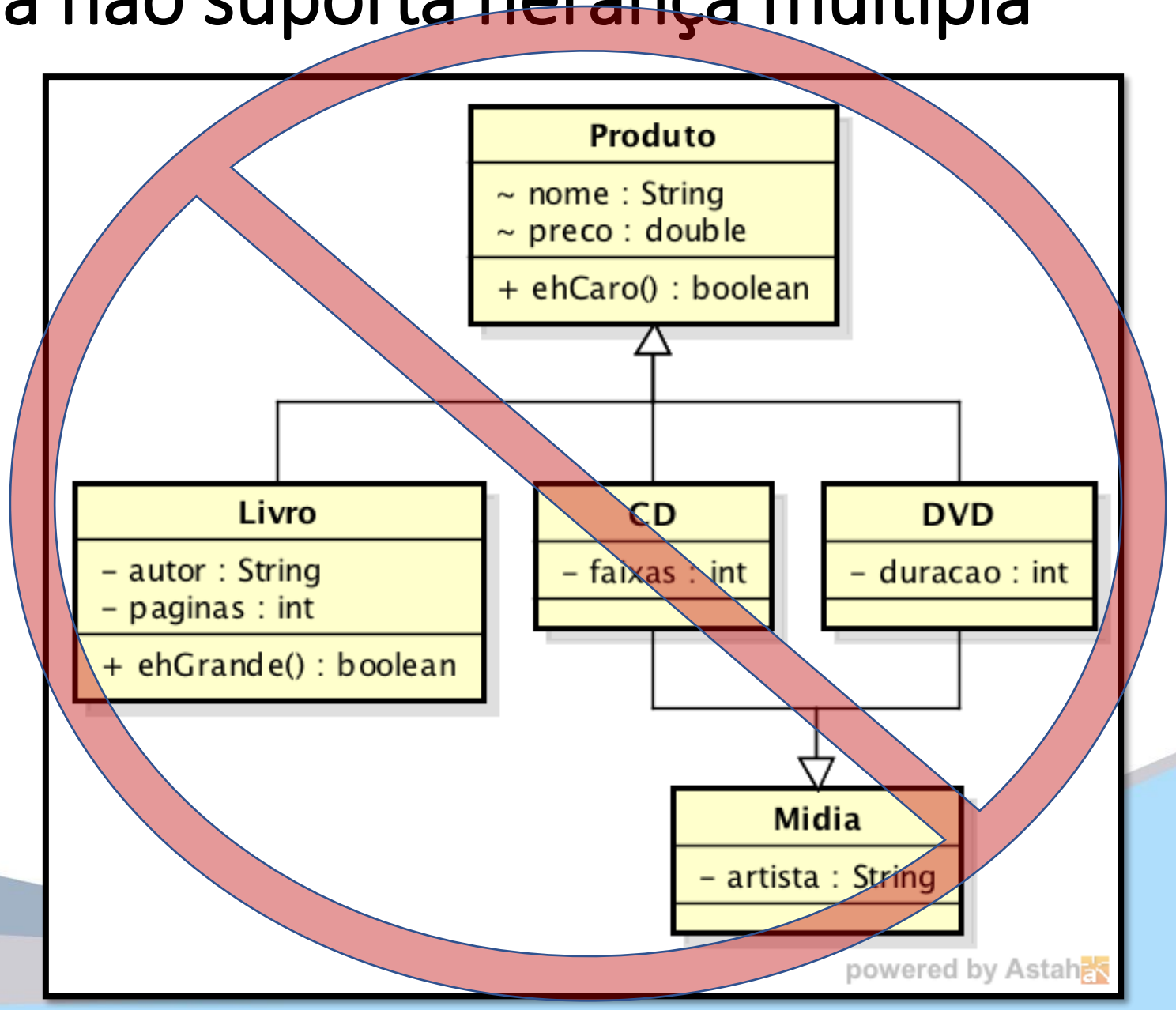
- Subclasse;
- Classe derivada;
- Classe filha/descendente, etc.

Java suporta herança simples

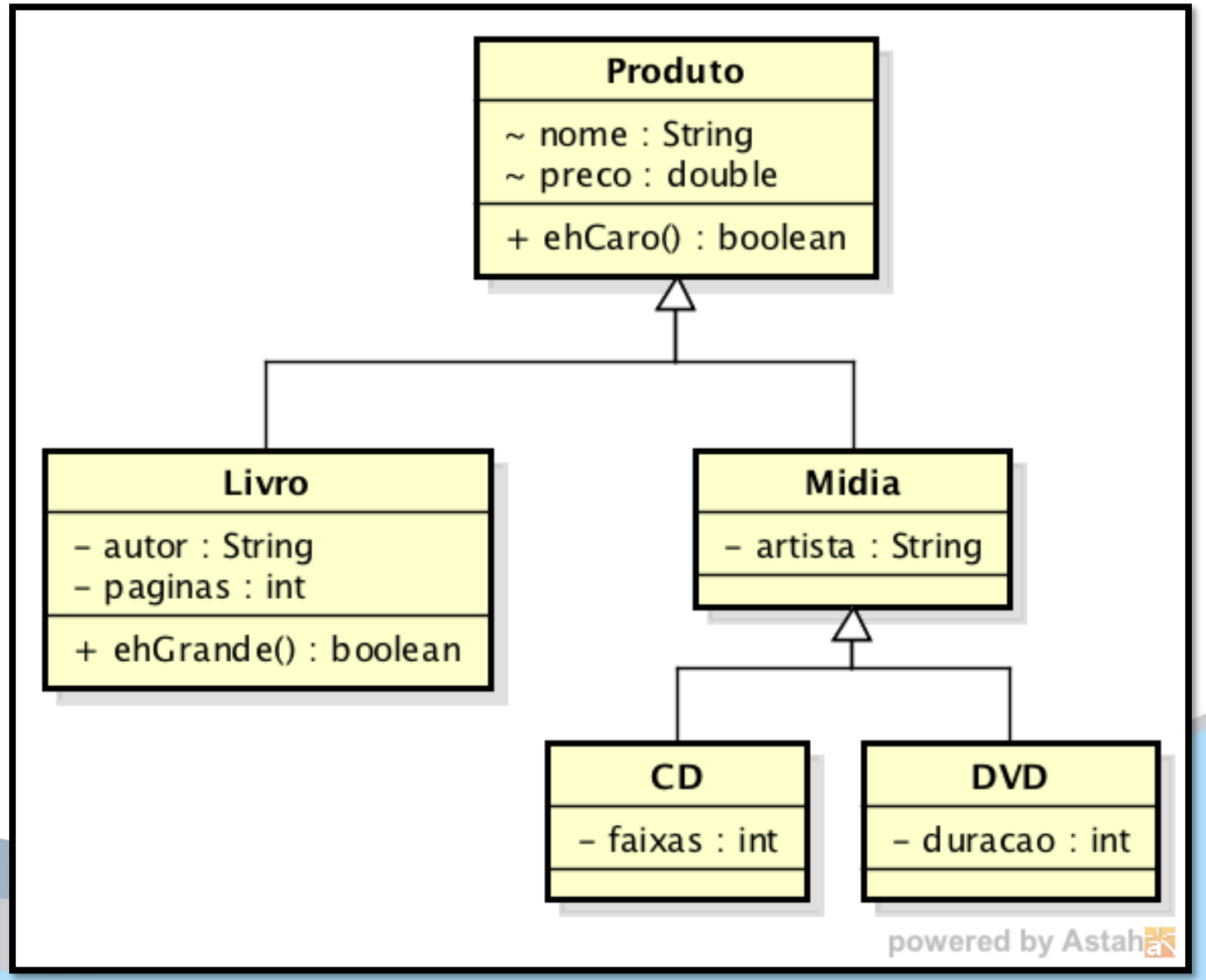
- Uma classe pode ter muitas subclases;
- Uma classe só pode ter uma superclasse.



Java não suporta herança múltipla



Hierarquias de qualquer tamanho



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

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.*

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Subclasses herdam membros

- 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 privativos?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Sobrescrita/Reescrita (Overriding)

- Quando uma subclasse implementa um método com a **mesma assinatura** de um método de uma superclasse com objetivo de mudar o seu funcionamento.
- Ex.: Considere a classe Funcionario com métodos para calcular salário que vimos anteriormente. Professor é um funcionário que recebe retribuição por titulação além dos demais valores. Como implementar?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Reescrita/sobrescrita de método

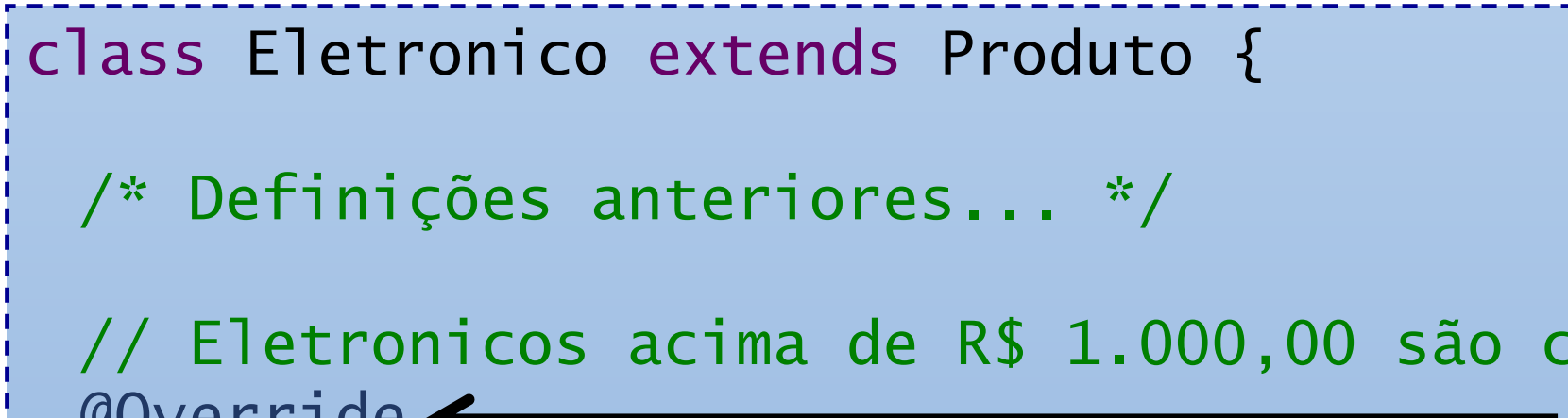
- Um método herdado pode não fazer total sentido:

```
public class Loja {
  public static void main(String[] args) {
    Eletronico tv = new Eletronico("TV 40\"", 200.0);

    // TV 40 polegadas por R$ 200? Uma pechincha!
    System.out.println(tv.ehCaro()); // true (??)
  }
}
```

Reescrita/sobrescrita de método

- 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?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

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.

Quanto mais cedo detectamos erros, melhor!

Polimorfismo

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Polimorfismo – Uso na Prática

Chamada de funções e **criação de estruturas de dados** capazes de armazenar objetos de várias classes diferentes, mas que possuem a mesma interface (métodos públicos).

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

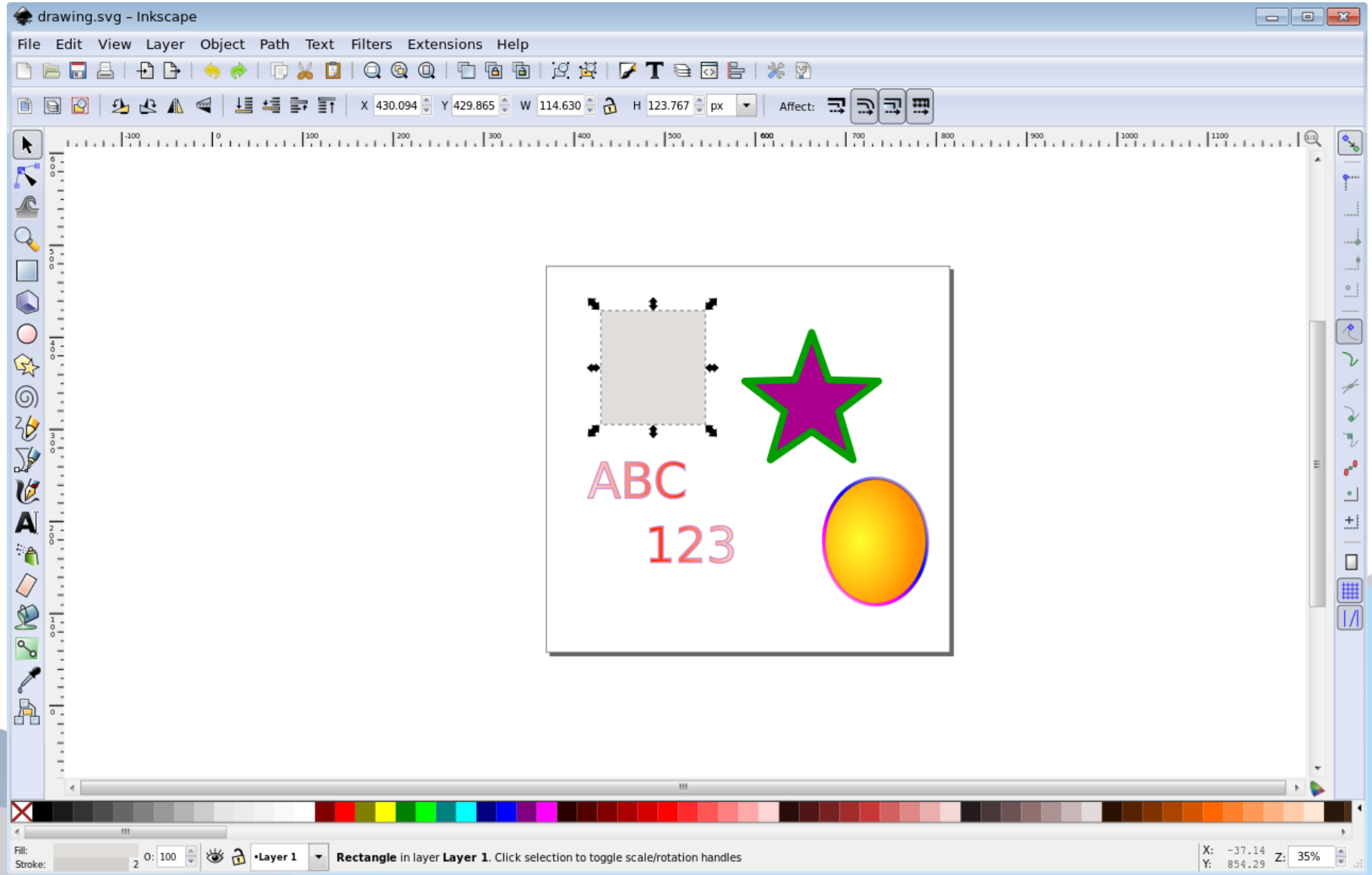
Polimorfismo

- 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



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Exemplo: um aplicativo de desenho

```
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 { /* ... */ }
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

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);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Ampliação

- 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!
  }
}
```

Incrementando o exemplo

- 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();
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Esquecendo o tipo do objeto

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

```
Forma f = new Quadrado();
```

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

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Métodos mais gerais

- 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?

```
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()`?

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

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.*

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

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.

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Benefícios do polimorfismo

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

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

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Benefícios do polimorfismo

```
class Reta extends Forma {
  @Override
  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);
  }
}
```

UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Benefícios do polimorfismo

- 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.

Exemplo - 1

- Crie uma classe abstrata Investimento com um atributo investimento_inicial que possa ser acessado apenas por suas subclasses e um método público e abstrato retorno(): double.
- Crie uma subclasse Poupança que tenha os atributos privados dias de investimento e taxa de lucro. Implemente o retorno como a soma dos lucros mensais, i.e., a cada mês será aplicada a taxa de lucro sobre o total parcial (juros compostos) e os valores devem ser somados.
- Crie uma subclasse Imovel que tenha como atributos a área do imóvel, o valor atual do m2, o valor do aluguel que o dono recebe mensalmente e o número de meses desde a compra. Implemente o método retorno como a soma da valorização e dos ganhos com aluguéis.
- Implemente os construtores de todas as classes.

```java
package src;

public abstract class Investimento {
    double investimento_inicial;

    public Investimento(double investimento_inicial) {
        this.investimento_inicial = investimento_inicial;
    }

    public abstract double retorno();
}
```

```java
package src;

public class Poupanca extends Investimento {
    private double taxa;
    private int n_dias;

    Poupanca(double investimento_inicial, double taxa, int n_dias) {
        super(investimento_inicial);
        this.taxa = taxa;
        this.n_dias = n_dias;
    }

    public double retorno() {
        int n_meses = n_dias / 30;
       return investimento_inicial * (Math.pow(1 + taxa, n_meses) - 1);
    }
}
```

```java
package src;

public class Imovel extends Investimento {

    private int tamanho;
    private double valor_m2;
    private double aluguel;
    private int meses_aluguel;

    public Imovel(double investimento_inicial,
                  int tamanho, double valor_m2, double aluguel, int meses_aluguel) {
        super(investimento_inicial);
        this.tamanho = tamanho;
        this.valor_m2 = valor_m2;
        this.aluguel = aluguel;
        this.meses_aluguel = meses_aluguel;
    }

    public double retorno() {
        double valorizacao = tamanho * valor_m2 - investimento_inicial;
        double ganho_aluguel = aluguel * meses_aluguel;
        return valorizacao + ganho_aluguel;
    }
}
```

```java
package src;

public class Programa {
    public static void main(String[] args) {
        Poupanca p = new Poupanca(1000, 0.008, 360);
        Imovel i = new Imovel(250000, 68, 7974, 1200, 36);

        System.out.println(p.retorno());
        System.out.println(i.retorno());
    }
}
```

Exemplo - 2

Implemente um método que retorne o maior retorno que pode ser obtido a partir de um ArrayList de Investimentos.

```
double buscaMelhorRetorno(ArrayList<Investimento> ivs) {
        double melhor_retorno = ivs.get(0).retorno();

        for (Investimento i : ivs) {
            double retorno_i = i.retorno();
            if (retorno_i < melhor_retorno) {
                melhor_retorno = retorno_i;
            }
        }

        return melhor_retorno;
    }
```