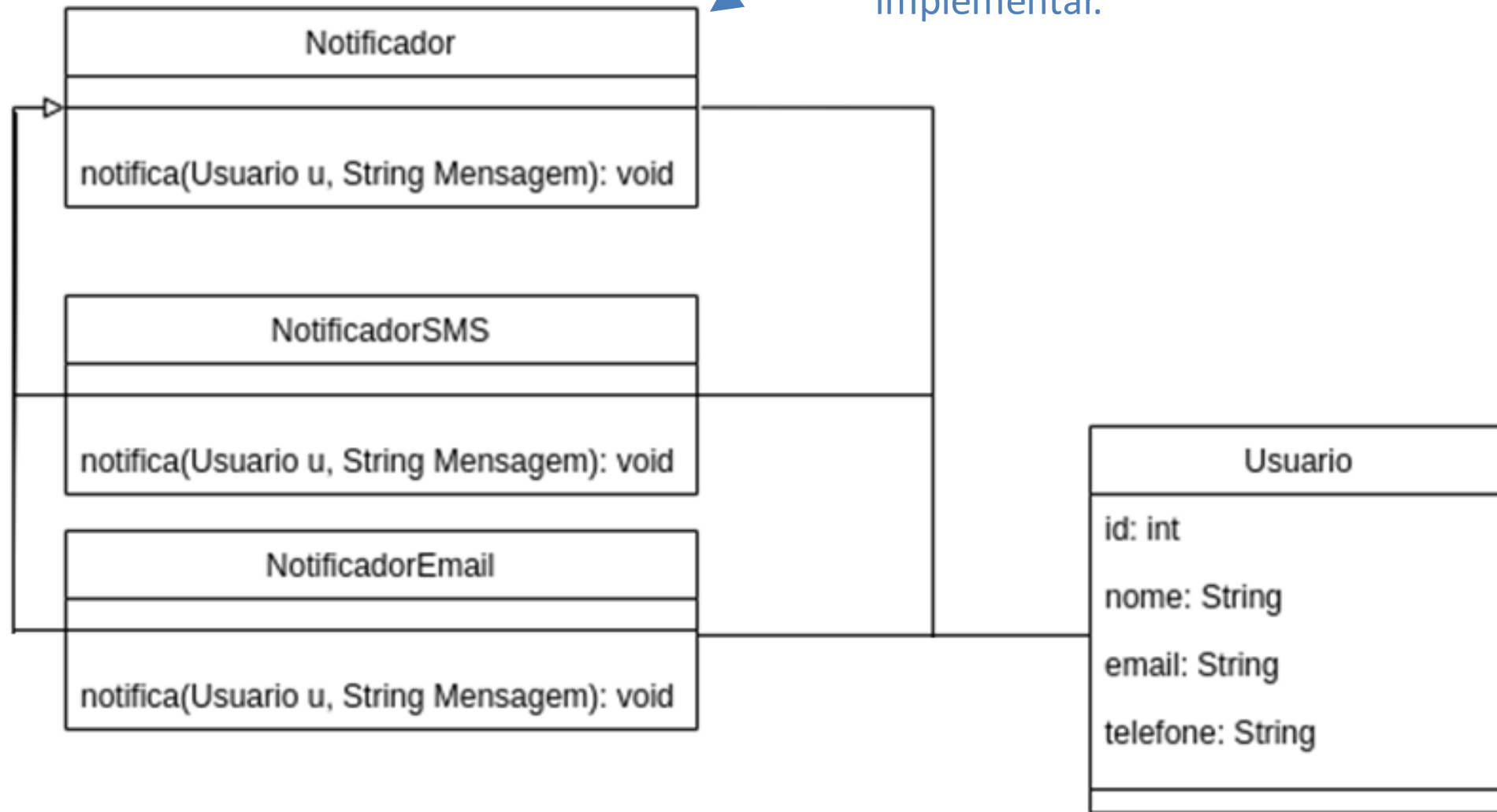




CLASSES ABSTRATAS E INTERFACES

A superclasse não sabe como enviar notificações. Ela só define o método que as subclasses “especializadas” devem implementar.

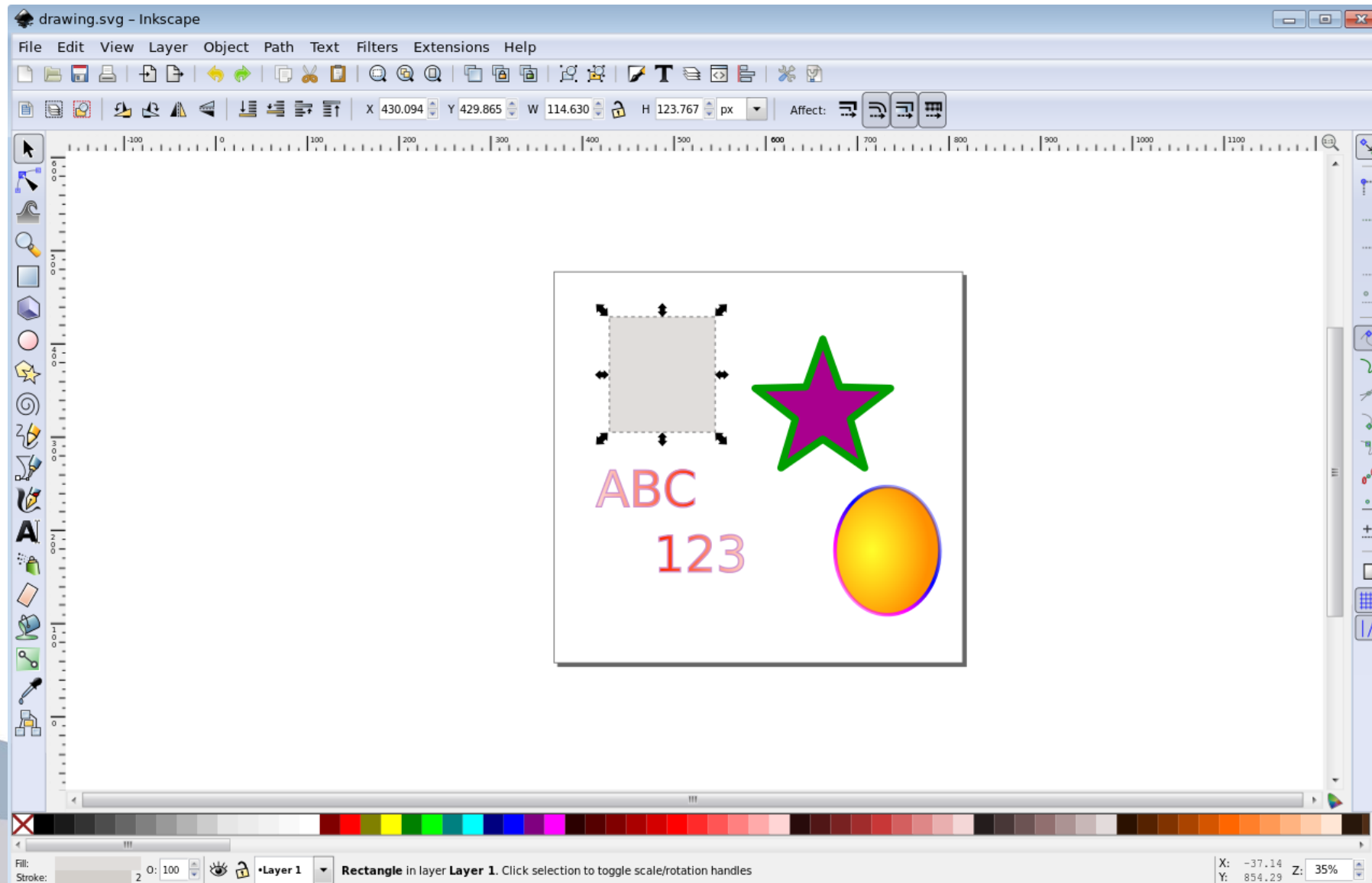


Uma solução possível

```
public class Notificador {
    void notifica(Usuario u, String mensagem) {}
}
```

```
public class NotificadorEmail extends Notificador {
    @Override
    void notifica(Usuario u, String mensagem) {
        System.out.println(
            "E-mail enviado para " + u.email
            + ": " + mensagem
        );
    }
}
```

Exemplo: um aplicativo de desenho



Classes e métodos abstratos

- Classes no topo da hierarquia podem ser muito **gerais**:
 - *O que é uma forma?*
 - *Como se desenha uma forma?*
 - *Como se aumenta uma forma?*
- Tais operações não fazem **sentido**. Queremos apenas definir que elas **existam**, mas não **implementá-las**;
- A **solução**: métodos **abstratos**.

Uma solução possível

Note o “;”. Métodos abstratos não possuem implementação.



```
abstract class Notificador {
    abstract void notifica(Usuario u, String mensagem);
}
```

```
public class NotificadorEmail extends Notificador {
    @Override
    void notifica(Usuario u, String mensagem) {
        System.out.println(
            "E-mail enviado para " + u.email
            + ": " + mensagem
        );
    }
}
```

Classes e métodos abstratos

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

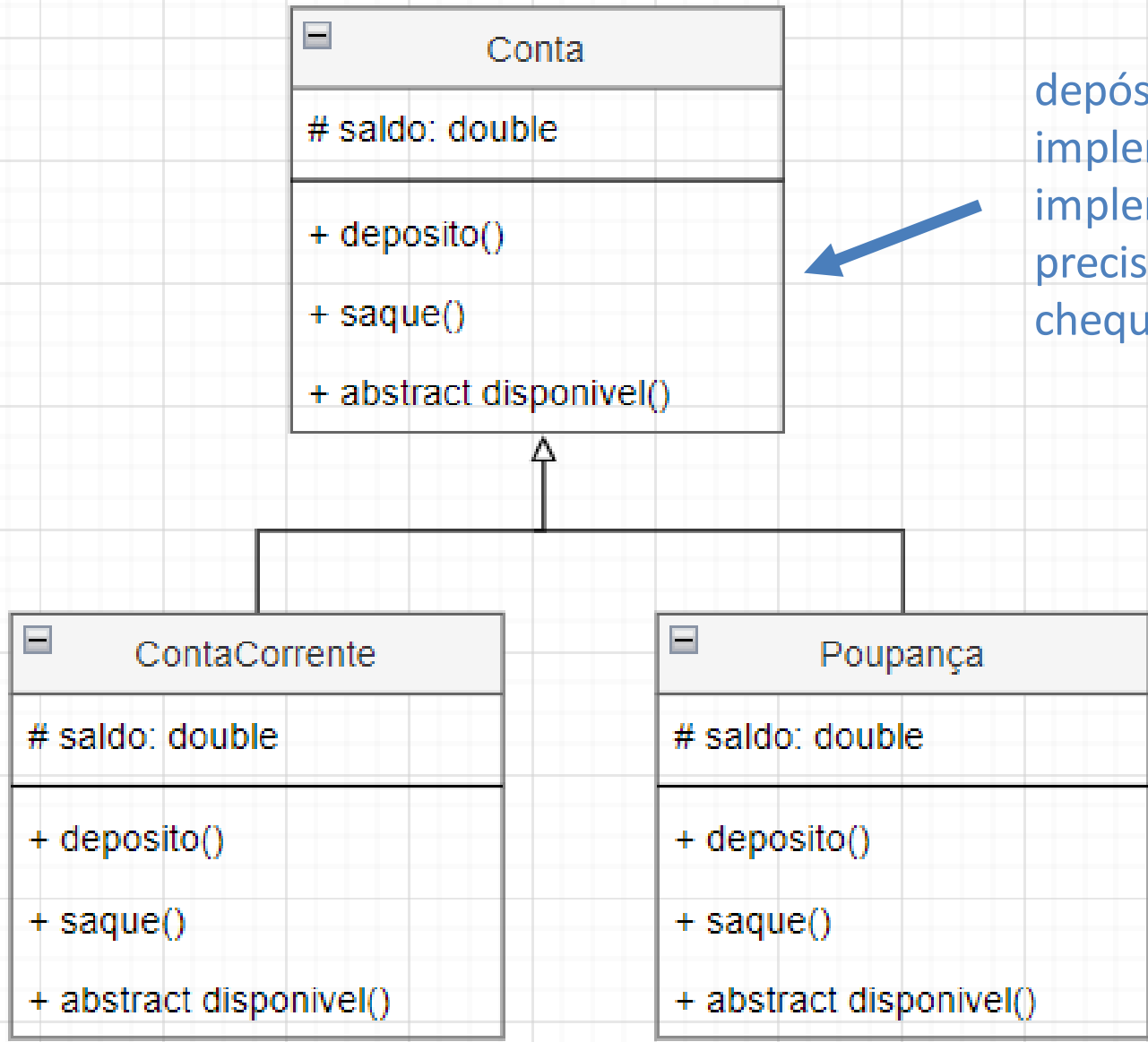
```
abstract class Forma {
    public abstract void desenhar();
}

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

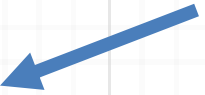
Classes abstratas

- 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.

Polimorfismo!



depósito e saque são implementados, mas para implementar o disponível, precisamos saber se existe cheque especial ou não.



```

public abstract class Conta {
    protected double saldo;

    public Conta(double saldo) {
        this.saldo = saldo;
    }

    public void deposito(double valor) {
        saldo += valor;
    }

    public abstract double disponivel();

    public void saque(double valor) {
        if (disponivel() >= valor)
            saldo -= valor;
        else
            System.out.println("Saldo insuficiente.");
    }
}

```

```
public class App {  
    public static void main(String[] args) {  
        // ERRO!!! Conta é abstrato.  
        Conta c = new Conta();  
    }  
}
```

```
public class ContaCorrente extends Conta {  
    private double limite;  
  
    public ContaCorrente(double saldo, double limite) {  
        super(saldo);  
        this.limite = limite;  
    }  
  
    @Override  
    public double disponivel() {  
        return saldo + limite;  
    }  
}
```

```
public class Poupanca extends Conta {
    public Poupanca(double saldo) {
        super(saldo);
    }

    @Override
    public double disponivel() {
        return saldo;
    }
}
```

```
public class App {
    public static void main(String[] args) {
        ContaCorrente c1 = new ContaCorrente(0, 200);
        Poupanca c2 = new Poupanca(0);

        c1.saque(100); // OK! Saque realizado!
        c2.saque(100); // Saldo insuficiente.
    }
}
```

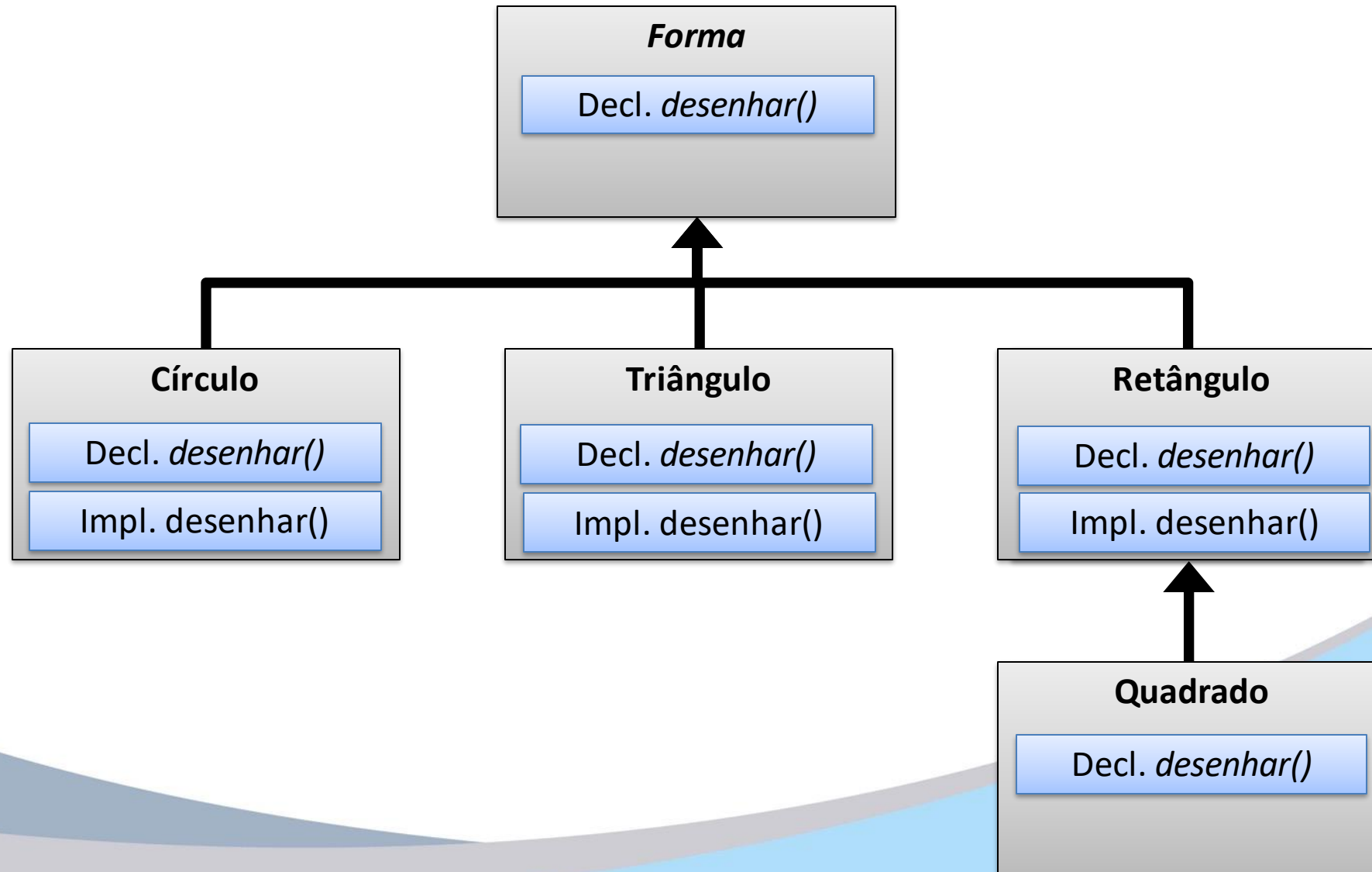
Com
polimorfismo,
tudo continua
funcionando.



```
public class App {
    public static void main(String[] args) {
        Conta c1 = new ContaCorrente(0, 200);
        Conta c2 = new Poupanca(0);

        c1.saque(100); // OK! Saque realizado!
        c2.saque(100); // Saldo insuficiente.
    }
}
```

Construindo a hierarquia de formas



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”.

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) {
        /* ... */
    }
}

```

Tudo é público na interface

- Métodos definidos na interface são **automaticamente públicos e abstratos**;
- Atributos definidos na interface são **automaticamente públicos e estáticos**.

Exceção: default methods (Java 8)

```
interface Forma {
    int x = 10;
    void desenhar();
}
```

```
interface Forma {
    public static int x = 10;
    public abstract void desenhar();
}
```

Definições
equivalentes

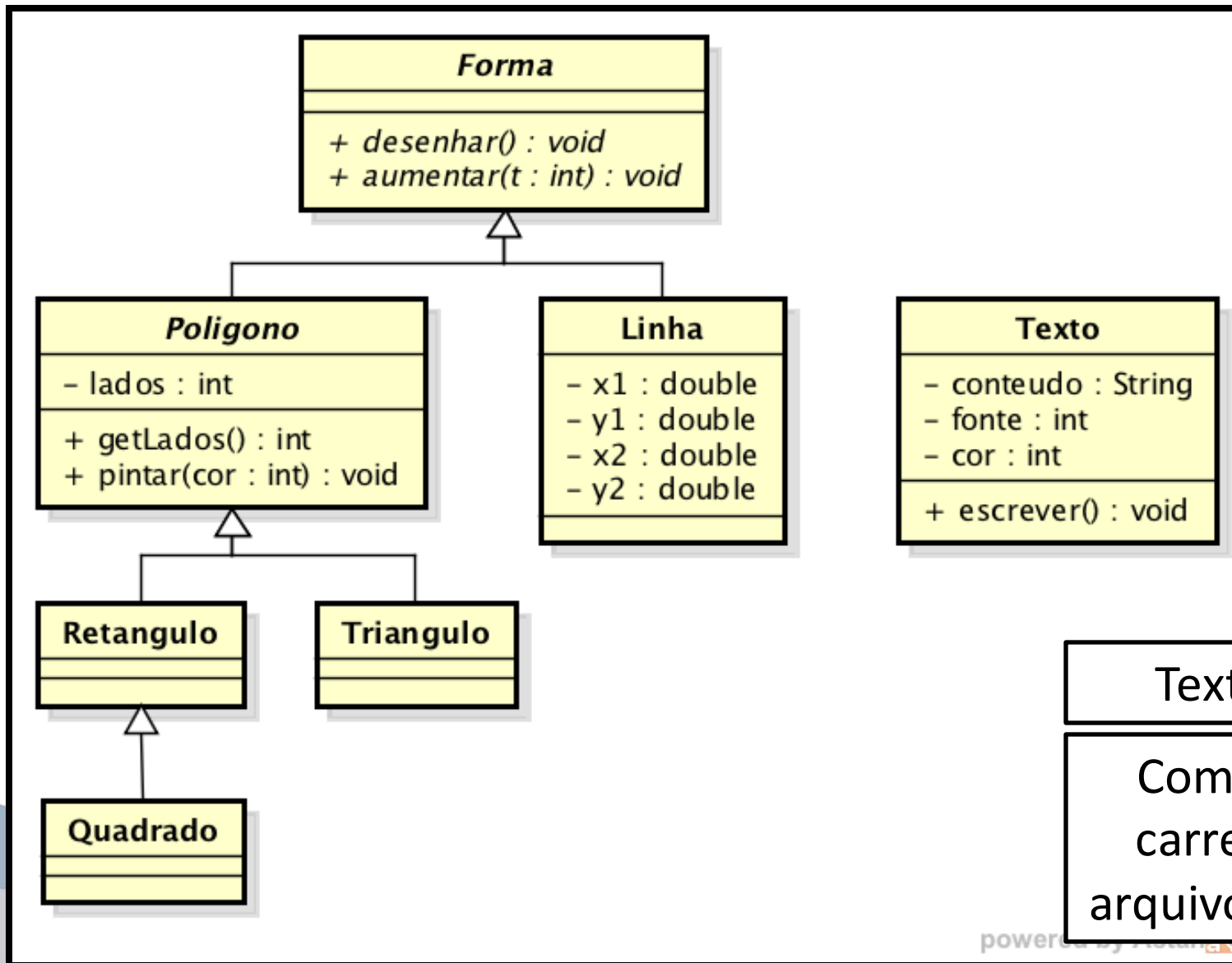
Por isso, cuidado com erros

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

class Linha implements Forma {
    // Erro: reduziu de público para priv. ao pacote!
    void desenhar() {
        /* ... */
    }

    // Erro: reduziu de público para privativo!
    private void aumentar(int t) {
        /* ... */
    }
}
```

Motivação



Texto não é forma, OK!

Como fica o método que
carrega meu desenho do
arquivo que eu salvei no HD?

Não-solução 1

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

Texto não é forma! Esse método não serve...

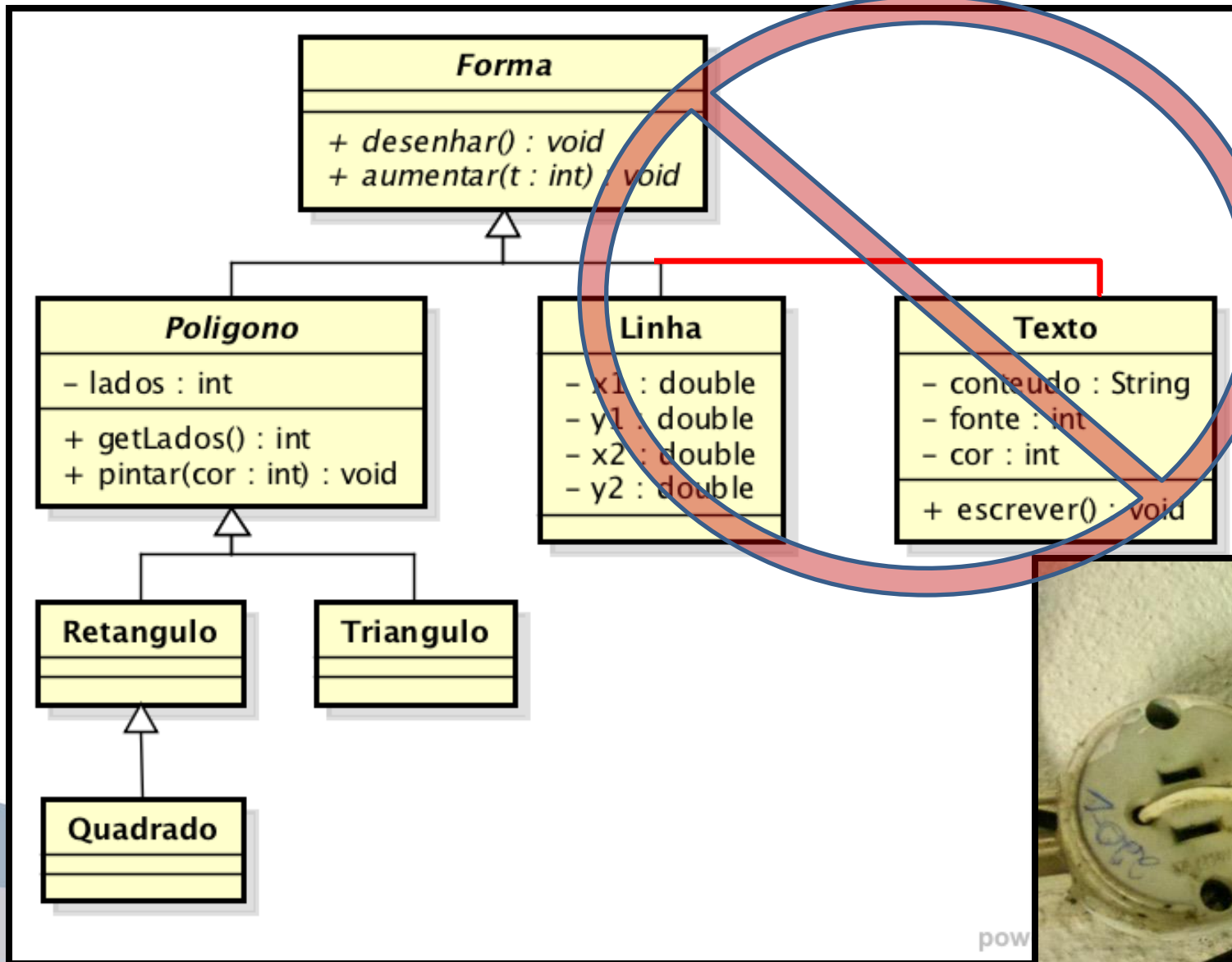
Não-solução 2

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

    private static void desenhar(Texto[] ts) {
        for (int i = 0; i < ts.length; i++)
            ts[i].escrever();
    }
}
```

Já vimos, com polimorfismo,
que essa não é a solução...

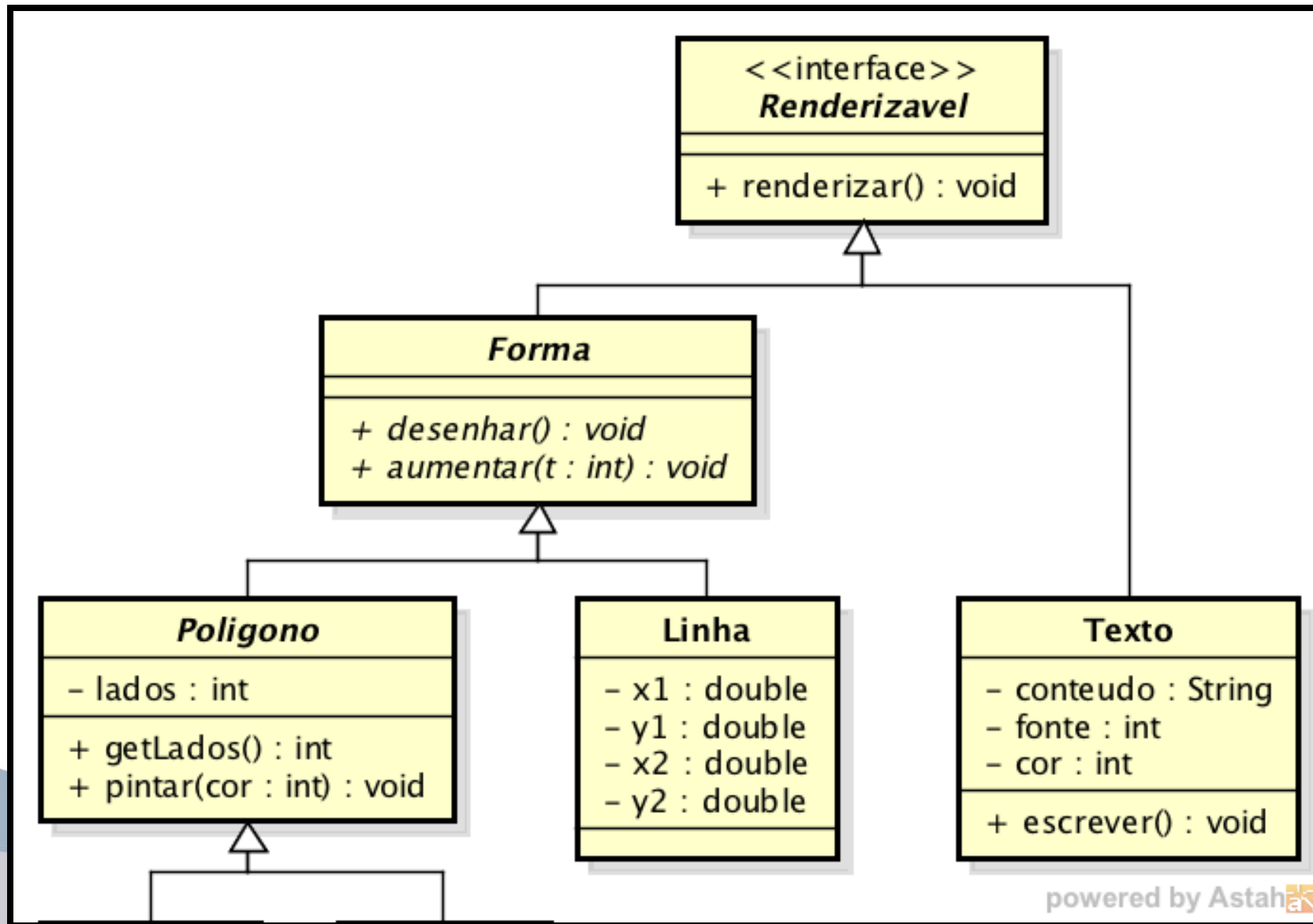
Não-solução 3



Já sei!



Solução: interfaces



Solução: interfaces

```
abstract class Forma implements Renderizavel {
    /* ... */
    @Override
    public void renderizar() {
        desenhar();
    }
}
```

```
class Texto implements Renderizavel {
    /* ... */
    @Override
    public void renderizar() {
        escrever();
    }
}
```

Solução: interfaces (alternativa)

```
interface Forma extends Renderizavel {
    /* ... */

    // As diferentes implementações de forma agora tem
    // que implementar renderizar() e não desenhar().
}

class Texto implements Renderizavel {
    /* ... */

    @Override
    public void renderizar() {
        escrever();
    }
}
```

Solução: interfaces

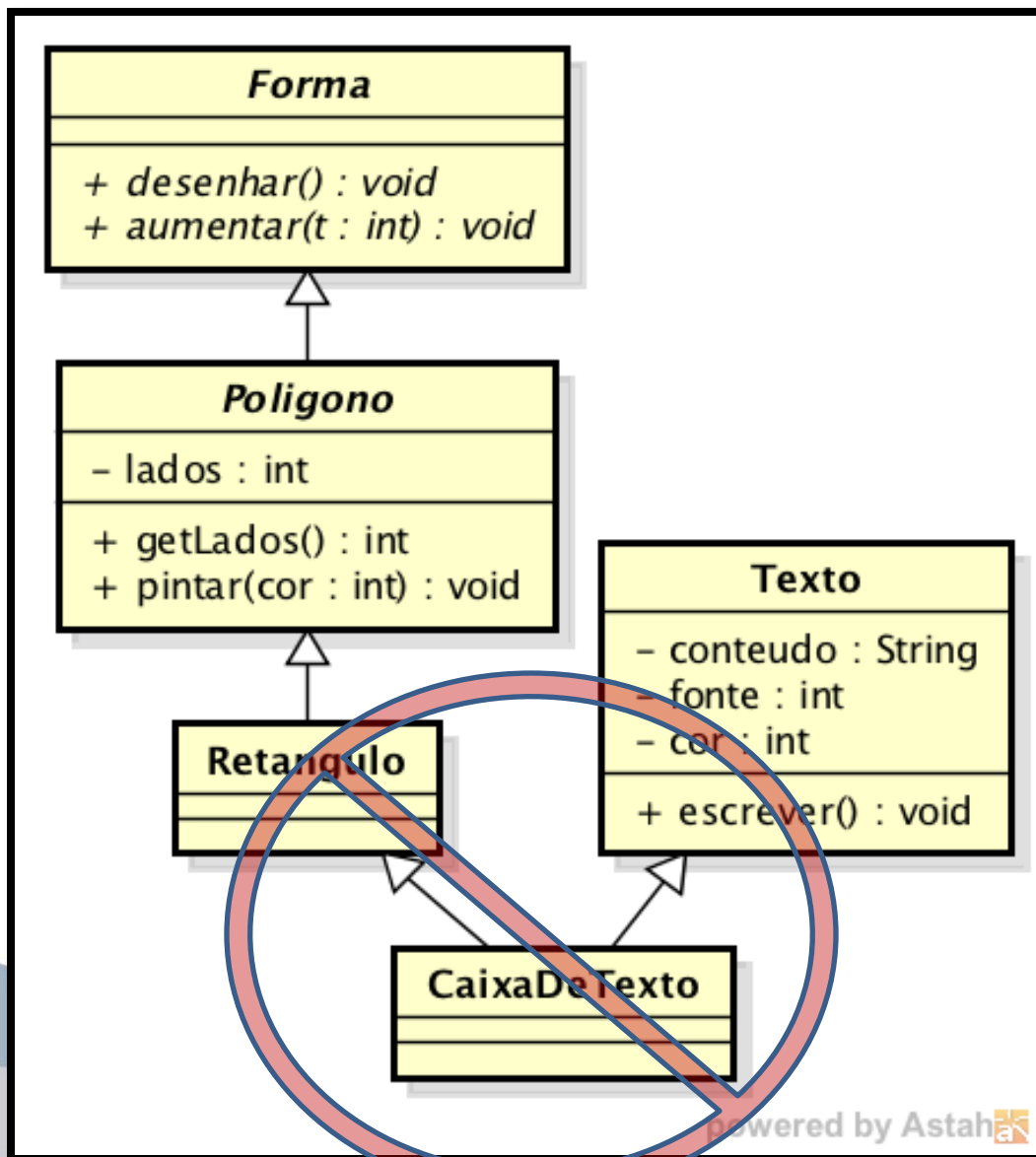
- O **polimorfismo** se amplia: mais um **modo** de **referenciar** uma forma: `Renderizavel`;
- `AplicativoDesenho` não precisa saber a **classe real** do objeto, apenas que ele **implementa** a **interface**;
 - *Se implementa a interface, ele **implementa** o método `renderizar()`!*
- **Novas classes** podem ser renderizáveis!

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

Interface = contrato

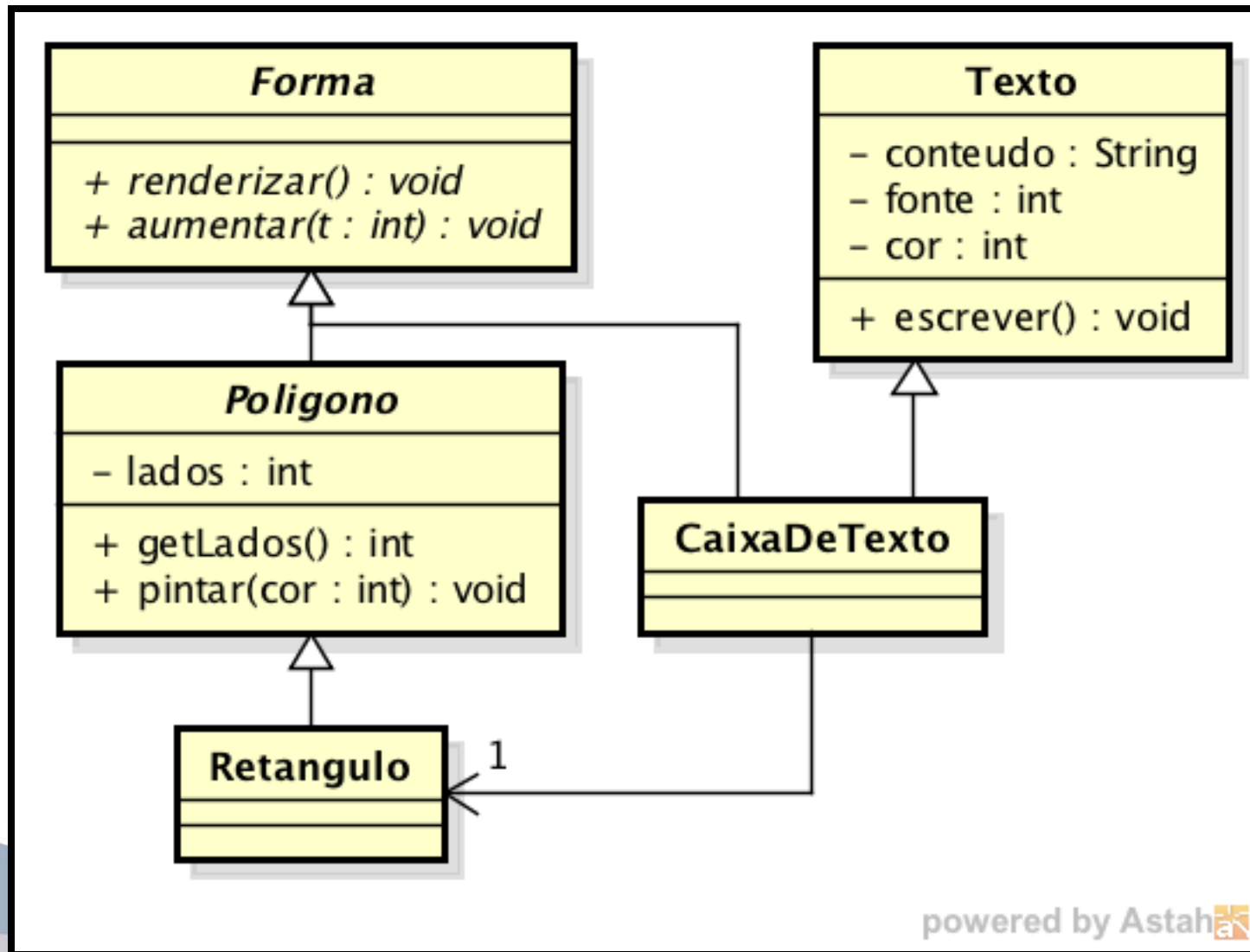
- Renderizavel define que **todas** as **classes** que a implementam **saibam** se renderizar() – “o **que**”;
 - *A implementação define “o como”;*
- Define um **contrato**: “quem desejar ser renderizável precisa saber se renderizar()”;
 - *A classe que quiser, **assina** o **contrato** e se responsabiliza a **cumpri-lo**;*
- Programe voltado a **interfaces** e não a **implementações** (mas sem **exageros**).

Herança múltipla em Java



Pode isso, Arnaldo?

Herança múltipla em Java



A regra é clara...

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();
    }
}

```

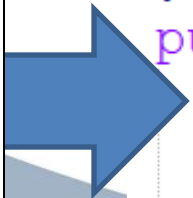
Classes Abstratas e Interfaces impõe Regras de Desenvolvimento

```

class Notificador {
    public void notifica(Usuario u) {}
}
public class NotificadorSMS extends Notificador {
    public void notifica(Usuario u) {
        System.out.println("SMS enviado para " + u.telefone);
    }
}
public class NotificadorEmail extends Notificador {
    public void notifica(Usuario u) {
        System.out.println("E-mail enviado para " + u.email);
    }
}
public class NotificadorPopUp extends Notificador {
    public void enviaNotificacaoPopup(Usuario u) {
        System.out.println("Pop-Up enviado para " + u.telefone);
    }
}

```

alguém
não
seguiu o
padrão



Funciona?

```
import java.util.ArrayList;

public class App {
    public static void main(String[] args) {
        ArrayList<Notificador> notificadores = new ArrayList<>();

        notificadores.add(new NotificadorSMS());
        notificadores.add(new NotificadorEmail());
        notificadores.add(new NotificadorPopUp());

        Usuario u = new Usuario("123", "Jose", "123-123", "jose@email.com");

        for (Notificador n : notificadores) {
            n.notifica(u);
        }
    }
}
```



Roda, mas não se comporta como esperado...
O método `enviaNotificacaoPopup` não é invocado

```
import java.util.ArrayList;
```

```
public class App {
    public static void main(String[] args) {
        ArrayList<Notificador> notificadores = new ArrayList<>();

        notificadores.add(new NotificadorSMS());
        notificadores.add(new NotificadorEmail());
        notificadores.add(new NotificadorPopUp());
```

```
    Usuario u = new Usuario("123", "Jose", "123-123", "jose@email.com");
```

```
    for (Notificador n : notificadores) {
        if (n instanceof NotificadorPopUp) {
            NotificadorPopUp p = (NotificadorPopUp) n;
            p.enviaNotificacaoPopUp(u);
        } else
            n.notifica(u);
    }
```

```
    }
}
```

**Até resolve, mas
imagine como a
complexidade (e
a bagunça)
aumentariam
com o tempo.**


Classes Abstratas e Interfaces impõe Regras de Desenvolvimento

```

abstract class Notificador {
    public abstract void notifica(Usuario u);
}
public class NotificadorSMS extends Notificador {
    public void notifica(Usuario u) {
        System.out.println("SMS enviado para " + u.telefone);
    }
}
public class NotificadorEmail extends Notificador {
    public void notifica(Usuario u) {
        System.out.println("E-mail enviado para " + u.email);
    }
}
public class NotificadorPopUp extends Notificador {
    public void enviaNotificacaoPopUp(Usuario u) {
        System.out.println("Pop-Up enviado para " + u.telefone);
    }
}

```

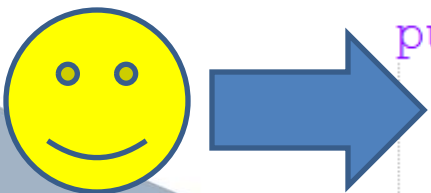
Error: The type NotificadorPopUp must implement the inherited abstract method Notificador.notifica(Usuario)Java(67109264)



Classes Abstratas e Interfaces impõe Regras de Desenvolvimento

```

abstract class Notificador {
    public abstract void notifica(Usuario u);
}
public class NotificadorSMS extends Notificador {
    public void notifica(Usuario u) {
        System.out.println("SMS enviado para " + u.telefone);
    }
}
public class NotificadorEmail extends Notificador {
    public void notifica(Usuario u) {
        System.out.println("E-mail enviado para " + u.email);
    }
}
public class NotificadorPopUp extends Notificador {
    public void notifica(Usuario u) {
        System.out.println("Pop-Up enviado para " + u.telefone);
    }
}
    
```



Classes Abstratas e Interfaces impõe Regras de Desenvolvimento

```
import java.util.ArrayList;

public class App {
    public static void main(String[] args) {
        ArrayList<Notificador> notificadores = new ArrayList<>();

        notificadores.add(new NotificadorSMS());
        notificadores.add(new NotificadorEmail());
        notificadores.add(new NotificadorPopUp());

        Usuario u = new Usuario("123", "Jose", "123-123", "jose@email.com");

        for (Notificador n : notificadores) {
            n.notifica(u);
        }
    }
}
```



Solução com Interfaces

```

interface Notificador {
    void notifica(Usuario u);
}
public class NotificadorSMS implements Notificador {
    public void notifica(Usuario u) {
        System.out.println("SMS enviado para " + u.telefone);
    }
}
public class NotificadorEmail implements Notificador {
    public void notifica(Usuario u) {
        System.out.println("E-mail enviado para " + u.email);
    }
}
public class NotificadorPopUp implements Notificador {
    public void notifica(Usuario u) {
        System.out.println("Pop-Up enviado para " + u.telefone);
    }
}

```

compact1, compact2, compact3
java.util

Class ArrayList<E>

```
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>
```



Hierarquia de Classes

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList



Interfaces



Class `AbstractList<E>`

```
java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
```

All Implemented Interfaces:

```
Iterable<E>, Collection<E>, List<E>
```

Direct Known Subclasses:

```
AbstractSequentialList, ArrayList, Vector
```

```
public abstract class AbstractList<E>
  extends AbstractCollection<E>
  implements List<E>
```

This class provides a skeletal implementation of the `List` interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array). For sequential access data (such as a linked list), `AbstractSequentialList` should be used in preference to this class.

To implement an unmodifiable list, the programmer needs only to extend this class and provide implementations for the `get(int)` and `size()` methods.

To implement a modifiable list, the programmer must additionally override the `set(int, E)` method (which otherwise throws an `UnsupportedOperationException`). If the list is variable-size the programmer must additionally override the `add(int, E)` and `remove(int)` methods.



Interface Cloneable

All Known Subinterfaces:

`AclEntry`, `Attribute`, `AttributedCharacterIterator`, `Attributes`, `CertPathBuilderResult`, `CertPathParameters`, `CertPathValidatorResult`, `CertSelector`, `CertStoreParameters`, `CharacterIterator`, `CRLSelector`, `Descriptor`, `GSSCredential`, `Name`

`public interface Cloneable`

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.

Invoking `Object`'s `clone` method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

By convention, classes that implement this interface should override `Object.clone` (which is protected) with a public method. See `Object.clone()` for details on overriding this method.

A interface Comparable

- Um **exemplo** de interface na API Java é a interface Comparable;
- Define o **método** compareTo(Object obj):
 - *Compara o objeto atual (this) com o objeto informado (obj);*
 - *Retorna 0 se this = obj;*
 - *Retorna um número negativo se this < obj;*
 - *Retorna um número positivo se this > obj.*
- Métodos **genéricos** a utilizam para **ordenar** coleções de elementos.

A interface Comparable

```

class Valor implements Comparable {
    int valor;

    public Valor(int v) { valor = v; }

    @Override
    public int compareTo(Object obj) {
        return valor - ((Valor)obj).valor;
    }

    @Override
    public String toString() {
        return "" + valor;
    }
}

```

A interface Comparable

```

public class Teste {
    static void imprimir(Object[] vetor) {
        for (int i = 0; i < vetor.length; i++)
            System.out.print(vetor[i] + "; ");
        System.out.println();
    }
    public static void main(String[] args) {
        Valor[] vetor = new Valor[] {
            new Valor(10), new Valor(3),
            new Valor(15), new Valor(7)
        };
        imprimir(vetor);    // 10; 3; 15; 7;
        Arrays.sort(vetor);
        imprimir(vetor);    // 3; 7; 10; 15;
    }
}

```


Tipos genéricos e Comparable

```
// Java 1.4:
class Pessoa implements Comparable {
    private String nome;
    public int compareTo(Object o) {
        Pessoa p = (Pessoa)o;
        return nome.compareTo(p.nome);
    }
}

// Com tipos genéricos:
class Pessoa implements Comparable<Pessoa> {
    private String nome;
    public int compareTo(Pessoa o) {
        return nome.compareTo(o.nome);
    }
}
```

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)
<code>int</code> para <code>long</code>	<code>long</code> para <code>int</code>
<code>float</code> para <code>double</code>	<code>double</code> para <code>float</code>
Cachorro para <code>Animal</code>	<code>Animal</code> para Cachorro
Gato para <code>Animal</code>	<code>Animal</code> 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).*

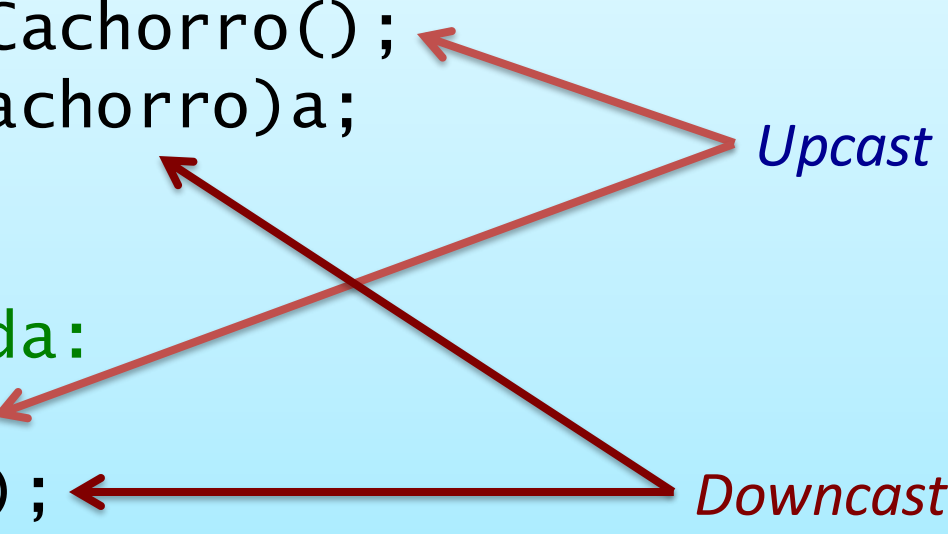
Upcast vs. downcast

```

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();
    }
}

```



Upcast

Downcast

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*.
- 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);
    }
}

```




UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO



UNIVERSIDADE FEDERAL
DO ESPÍRITO SANTO

Classes e métodos abstratos - Quiz

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

Exercitar é fundamental

- Apostila FJ-11 da Caelum – Classes Abstratas:
 - *Seção 10.6, página 145 (Exercícios: Classes Abstratas);*

- Apostila FJ-11 da Caelum – Interfaces:
 - *Seção 11.5, página 157 (Exercícios: Interfaces);*
 - *Seção 11.6, página 160 (Exercícios opcionais);*
 - *Seção 11.7, página 161 (Discussão: favoreça composição em relação à herança).*