

**SOU! CODE**

**PROGRAMAÇÃO ORIENTADA A  
OBJETOS**

# PROGRAMAÇÃO ORIENTADA A OBJETOS

SOUL CODE



Criar funcionalidades para  
a aplicação...



**ENTIDADES SÃO...**

**CLASSES**



**Abstrações**

# Podemos ter a classe GATO

(é uma abstração, pois podem existir vários tipos de gatos), apesar de todos os gatos se encaixarem na classe, cada tem suas características.

## Atributos:

- raça
- idade
- nome
- cor da pelagem
- vacina

## Métodos:

- vacinar
- dar banho
- castrar

# OBJETO

É uma instância concreta da classe.

CLASSE GATO

↓  
INSTÂNCIA

UM OBJETO GATO  
(objetos únicos)

# CLASSE

- Abstrata
- Instancia um objeto

# OBJETO

- Concreto
- É único

# CLASSES

## Atributos e Métodos

- Os métodos podem modificar os valores dos atributos
- Em Java, quase tudo que criamos é uma classe

# CLASSES

```
public class Gato{
```

...

}

# **CLASSES** – como um sistema pode ter muitas classes, elas costumam ser organizadas em pacotes.

**SOUL CODE**

```
import package org.animais;  
  
public class Gato{  
  
    // todas as informações que uma classe precisa saber devem ficar dentro dos atributos.  
    int idade;  
    String raca;  
  
}
```

# Criando objetos e acessando atributos

```
Gato otto = new Gato();  
Gato mingau = new Gato();
```

```
otto.raca = "vira lata";  
int id = otto.idade;
```

Os atributos podem ser acessados para leitura e escrita

# CLASSES

Retorno  
(void não retorna nada)

```
void vacinar(){  
    vacina = "nome da vacina";  
}
```

modifica o atributo vacina

pode ou não  
ter parâmetros

Para fazer a chamada do  
método:

```
otto.vacinar();
```

# Construtores

São “métodos especiais” usados para criar objetos da classe.

```
public class Gato{  
    String nome;  
    int idade;  
Gato(String nome){  
    this.nome = nome;  
    idade = 1;  
}  
}
```

Com eles você pode parametrizar o objeto criado ou inicializar variáveis.

“this” usado para referenciar elementos da classe

## Class Gato

Nome  
Idade

Fazer aniversário  
Tomar vacina

atributos  
métodos

Responsabilidades

## Objeto MeuGato

Nome = “Otto”  
Idade = 1 ano

Fazer aniversário  
Tomar vacina

atributos  
métodos

Responsabilidades

# Responsabilidades

**Tipo sabe: atributos: variáveis de instância**

**Tipo faz: comportamentos: métodos**

# Lógicas das Responsabilidades

## Exemplo: Classe Conta Corrente

Responsabilidades  
do tipo Sabe

- Sabe o número da conta
- Sabe o valor do saldo atual

Responsabilidades  
do tipo Faz

- Credita valor ao saldo atual
- Debita valor do saldo atua
- Verifica se é possível sacar valor

# Lógicas das Responsabilidades

## Exemplo: Classe Conta Corrente

### Responsabilidades do tipo Sabe

- atributo número da conta
- atributo saldo da conta

*Devolvem os valores dos atributos através do get e set (métodos assessores)*

### Responsabilidades do tipo Sabe

- Credita valor ao saldo atual
- Debita valor do saldo atua
- Verifica se é possível sacar valor

*Manipulam os valores dos atributos através de métodos próprios*

# Lógicas das Responsabilidades

## Exemplo: Classe Conta Corrente

*Gerar o código em java  
Com classe ContaCorrente e classe Principal  
para instanciar os objetos da classe.*

# Exercício

**Crie uma classe Banco com:  
3 atributos e 3 métodos**

# Classe colaboradora

**Agora é combinar as duas classes para que ela trabalhem em colaboração.**

**Classe Cliente**

depende

**Classe Servidora**

*Depende porque precisa da colaboração  
da classe servidora - Acoplamento.*

# Relacionamento entre Classes



As classes de um Sistema devem ter responsabilidades bem definidas e colaborar para implementar as funcionalidades Sistema!



## Pizza

Calcula o preço baseado nos ingredientes da pizza.



## Entrega

Calcula a entrega baseado no dia da semana e na distância.



## Carrinho

Calcula o valor total da compra com as pizzas e a entrega.

*A ideia real da orientação a objeto, que é combinar várias para gerar a funcionalidade de uma aplicação.*



São classes independentes que colaboram entre si.



**Criar um sistema orientado  
a objetos tem a ver com  
descobrir suas peças e  
como elas se encaixam!**

*Exemplo com um  
registrar de pontos.*

# Métodos e Atributos Estáticos

*Algumas classes possuem variáveis que são compartilhadas com todas as instâncias.*



**Classe**

*Instâncias possuem valores que são diferentes para cada uma delas.*



**Instâncias**

a variável precisa ter o modificador static

```
public class Gato {  
    static int totalGatos = 0;  
  
    Gato () {  
        totalGatos++;  
    }  
}
```

o mesmo valor é compartilhado por qualquer instância

CUIDADO!

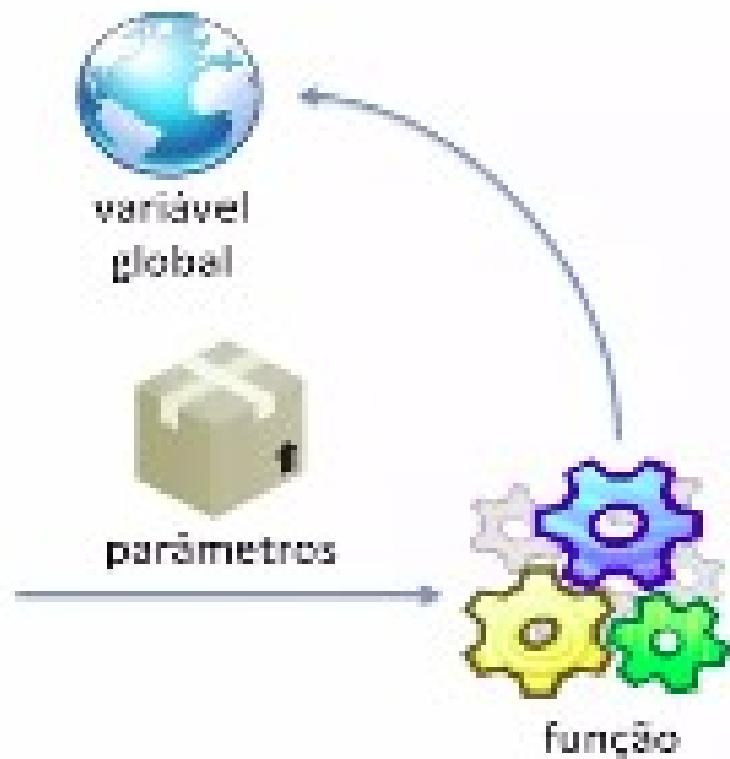
acelerar(carro);

**Pensando Estruturado**

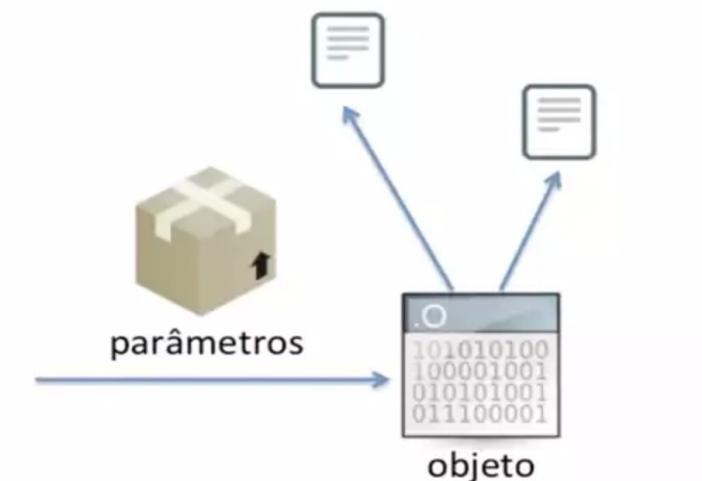
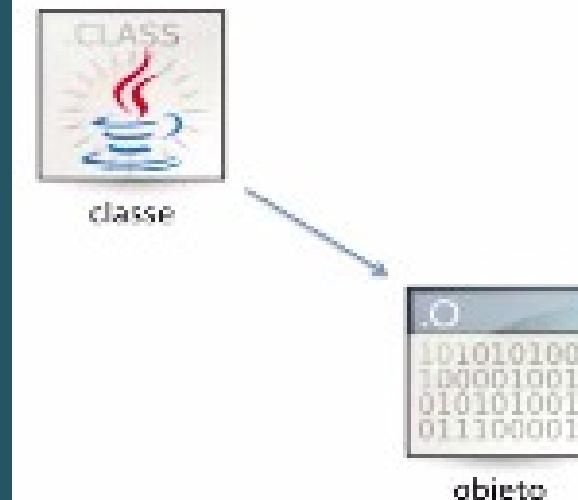
carro.acelerar();

**Pensando Orientado a Objetos**

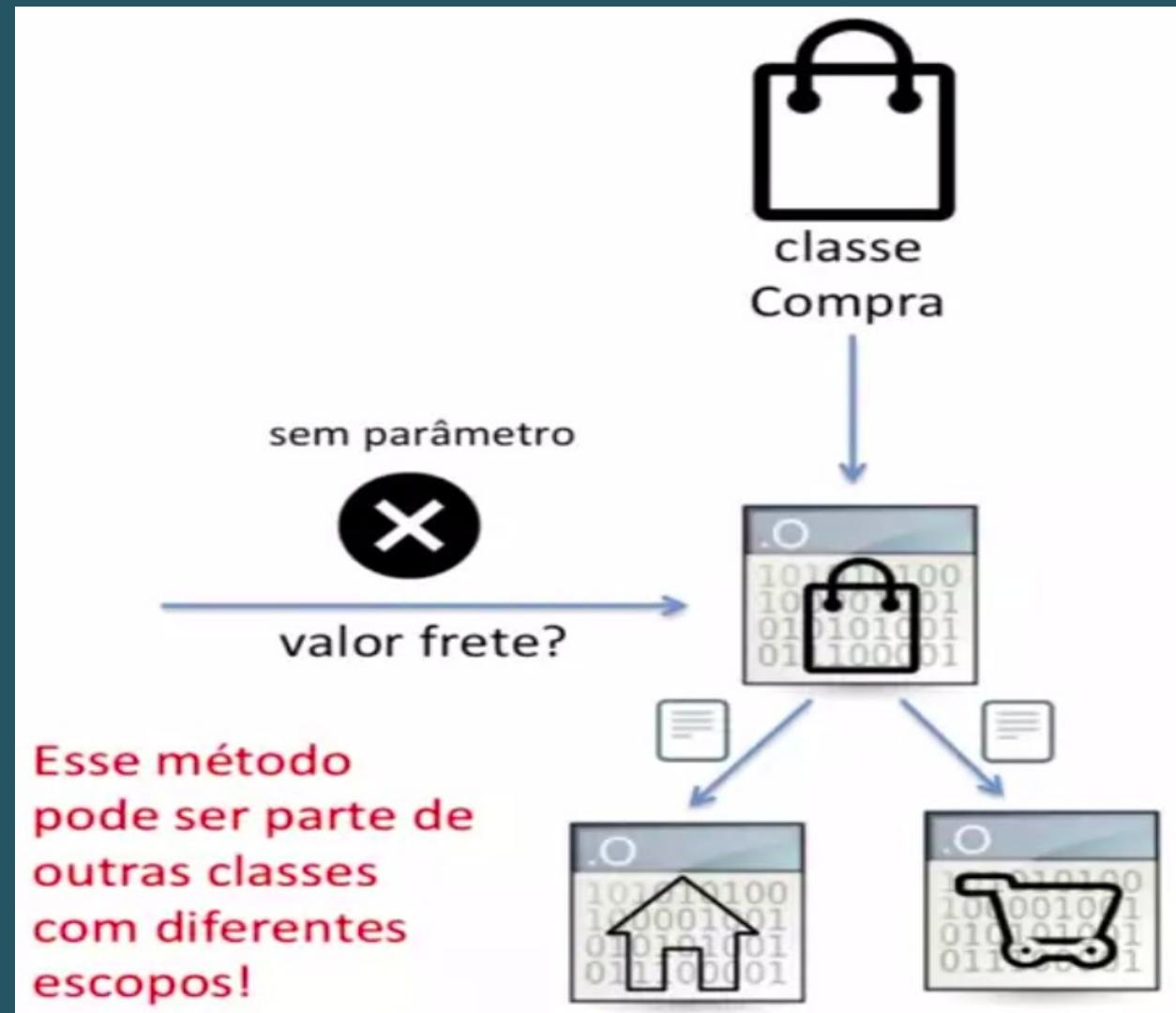
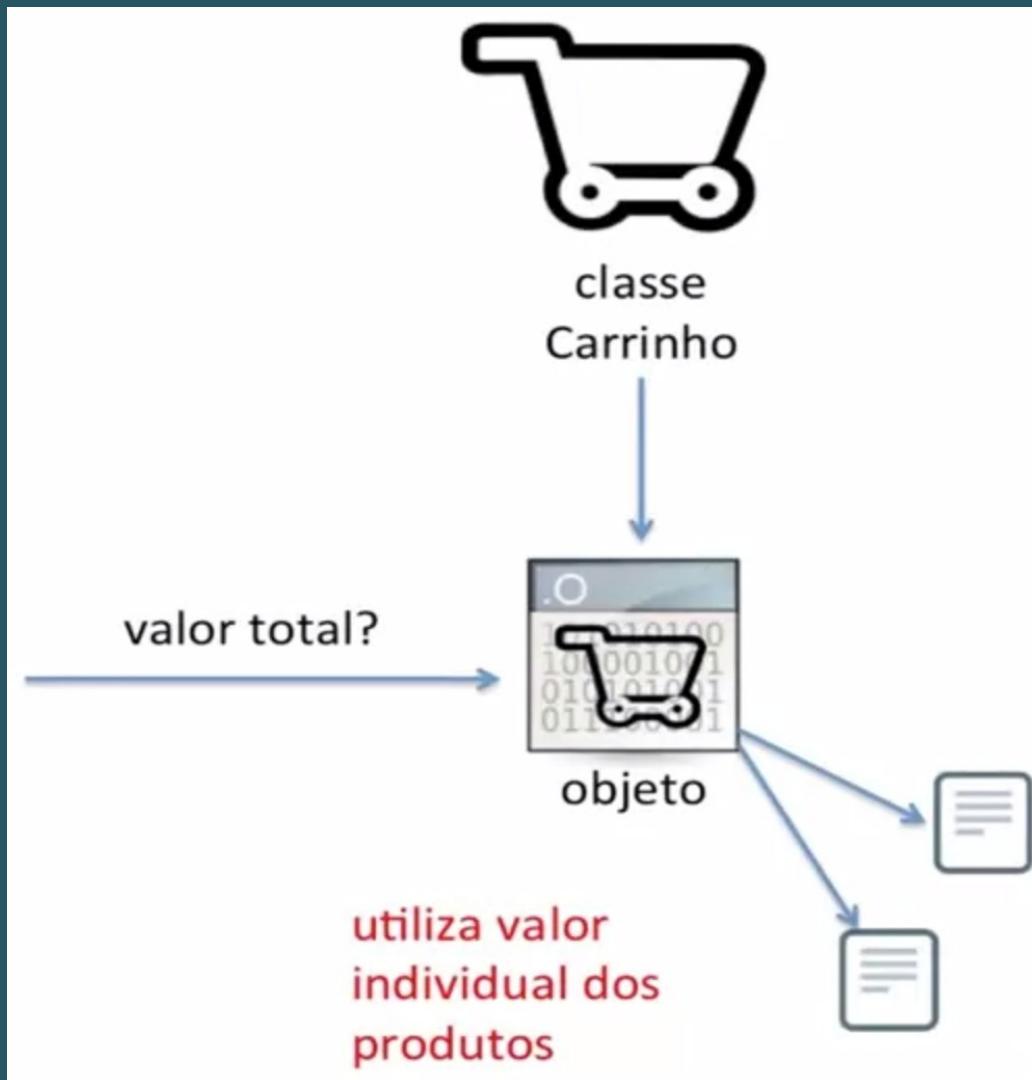
# Programação Estruturada



# Programação Orientada a Objetos



Um objeto executa seus métodos com base nos parâmetros recebidos e em seus dados internos!



# Modelagem CRC

## Passo 1 - Identificando Classes

Cartões CRC

Especificação do Sistema

**C – Classe**

**R – Responsabilidade**

**C – Colaboração**

- Identificar as Entidades
- Identificar o fluxo dos dados dentro da Aplicação
- Colocar a descrição de cada classe

# Modelagem CRC

## Passo 2 - Identificando Responsabilidades e colaborações

Cartões CRC

C – Classe

R – Responsabilidade

C – Colaboração

Responsabilidade

Para cada uma das classes:

- Responsabilidade do tipo Sabe
- Responsabilidade do tipo Faz

# Modelagem CRC

## Passo 4 – Construir a lógica de cada responsabilidade

Cartões CRC

C – Classe

R – Responsabilidade

C – Colaboração

Responsabilidade

Para cada uma das classes:

- Descreve o passo-a-passo de cada responsabilidade

# Modelagem CRC

## Passo 5 – Identificar as classes colaboradoras

Cartões CRC

C – Classe

R – Responsabilidade

C – Colaboração

Responsabilidade

Para cada uma das classes:

- Identificar de qual classe vem a colaboração

## Exemplo - SAB

A biblioteca tem nome, mantém catálogos de livros onde cada livro tem título, autor e número único de catálogo. Há usuários da biblioteca registrados, cada com nome único de usuário. O Usuário da biblioteca pode tomar livro para empréstimo e devolver o livro emprestado para o sistema. No entanto, cada transação, cada realização de empréstimo ou devolução de livro deve ser registrada por uma bibliotecária. A bibliotecária tem que registrar novo usuário da biblioteca no SAB. Quando solicitada, ela pode exibir, ordem crescente do número único de catálogo, os livros disponíveis para empréstimo, bem como exibir os livros já emprestados e, portanto, indisponíveis para empréstimo.

Classe: Biblioteca	
Responsabilidade	Colaboração
Registra usuário	Colaboradora: Usuario Colaboração: Construtor
Adiciona livro ao catálogo	Colaboradora: Livro Colaboração: Construtor
Empresta livro	Colaboradora: Livro e Usuario Colaboração: Anexa usuário e Anexa livro
Devolve livro	Colaboradora: Livro Colaboração: Desanexa livro e Desanexa usuário
Exibe livros disponíveis empréstimo	Colaboradora: Livro Colaboração: Sabe disponibilidade empréstimo
Exibe livros indisponíveis empréstimo	Colaboradora: Livro Colaboração: Sabe disponibilidade empréstimo



Classe: Livro	
Responsabilidade	Colaboração
Anexa usuário do empréstimo	Colaboradora: Colaboração:
Desanexa usuário do empréstimo	Colaboradora: Colaboração:

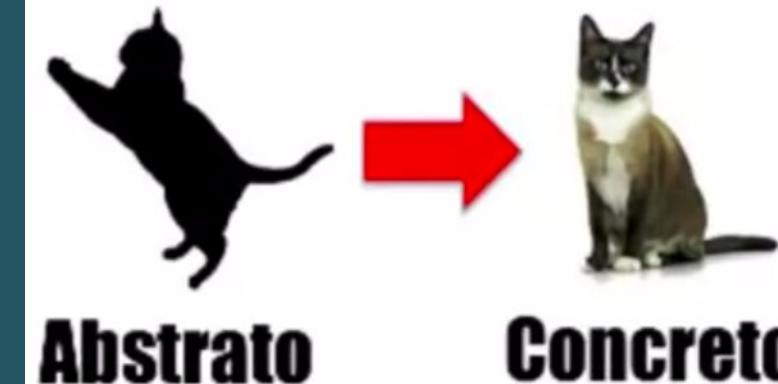




**Todos são  
abstrações!**

## Trabalhando com Abstrações

*Já entendemos como as classes são as abstrações a partir das quais são criados os objetos!*



# Exemplo – as duas classes de Empregados

SOUL CODE

```
public class Empregado{  
    String nome;  
    int idade;  
    double salario;  
}
```



```
public class Gerente{  
    String nome;  
    int idade;  
    double salario;  
    double bonus;  
}
```



```
public class Empregado{  
    String nome;  
    int idade;  
    double salario;  
}
```



**Duplicado!**

```
public class Gerente{  
    String nome;  
    int idade;  
    double salario;  
    double bonus;  
}
```

**Solução?**

O gerente é  
um tipo  
especial de  
empregado.

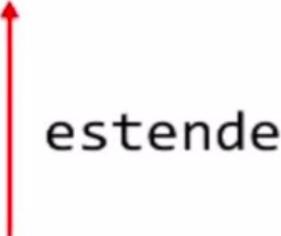
# Utilizar herança em uma modelagem

SOUL CODE

A partir da herança é possível estender uma classe, Adicionando atributos e modificando comportamento de métodos.



**SUPERCLASSE**



**SUBCLASSE**

**PODE**

```
public class Gerente  
    extends Empregado{  
    double bonus;  
}
```

- Adicionar métodos
- Adicionar atributos
- Modificar métodos

- Remover métodos
- Remover atributos
- Estender outra classe

```
public class Gerente  
    extends Empregado{  
    double bonus;  
}
```

**NÃO PODE**

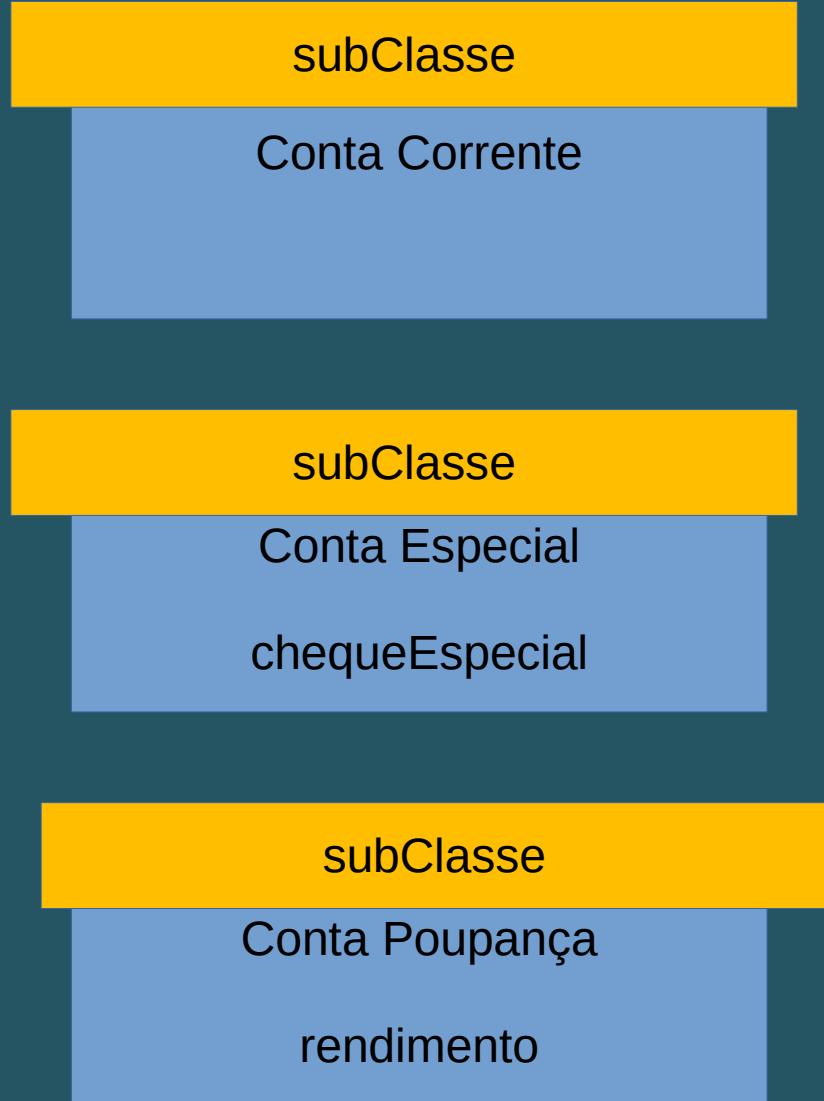
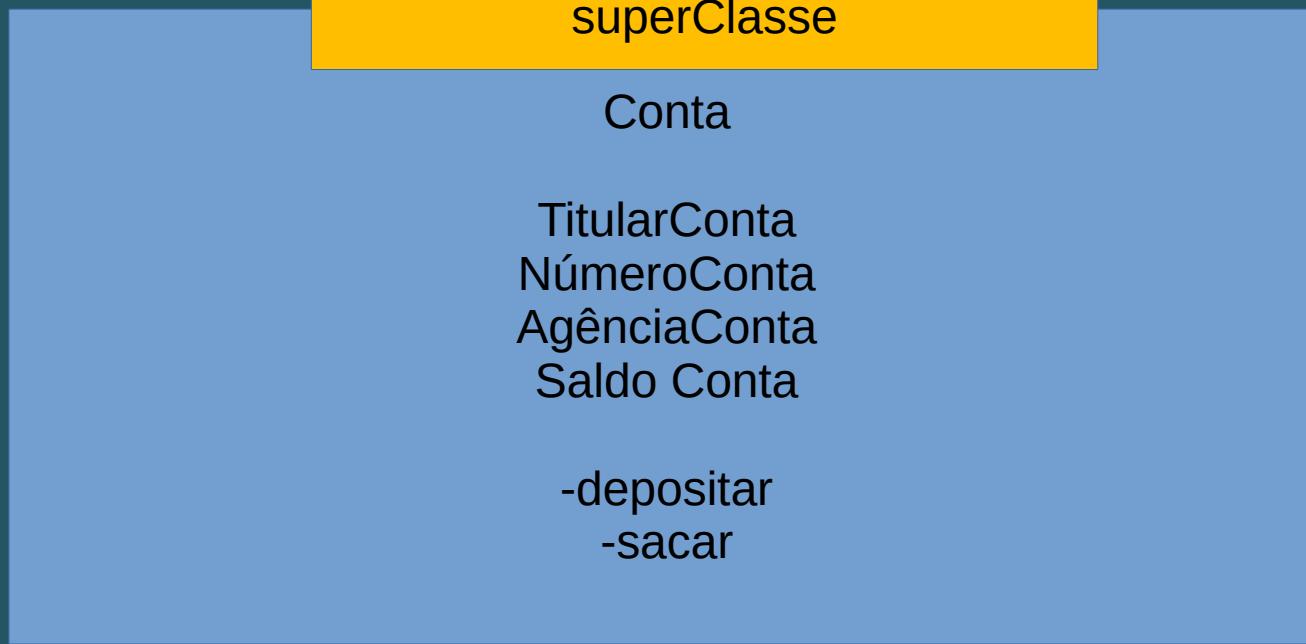
```
public class Empregado{  
    String nome;  
    int idade;  
    double salario;  
}
```



*herda métodos e atributos*

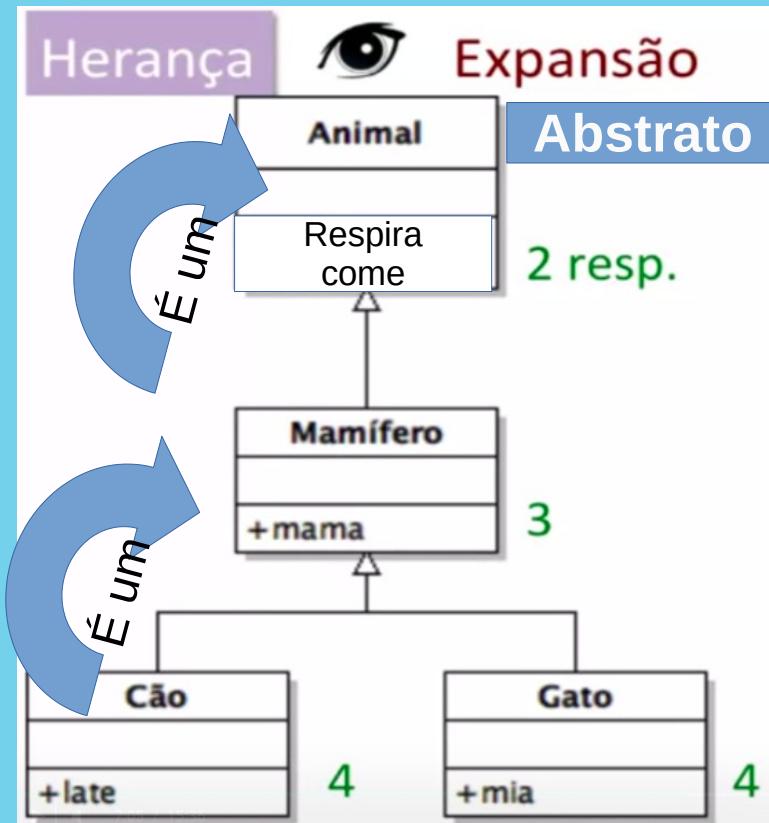
```
public class Gerente  
    extends Empregado{  
    double bonus;  
}
```



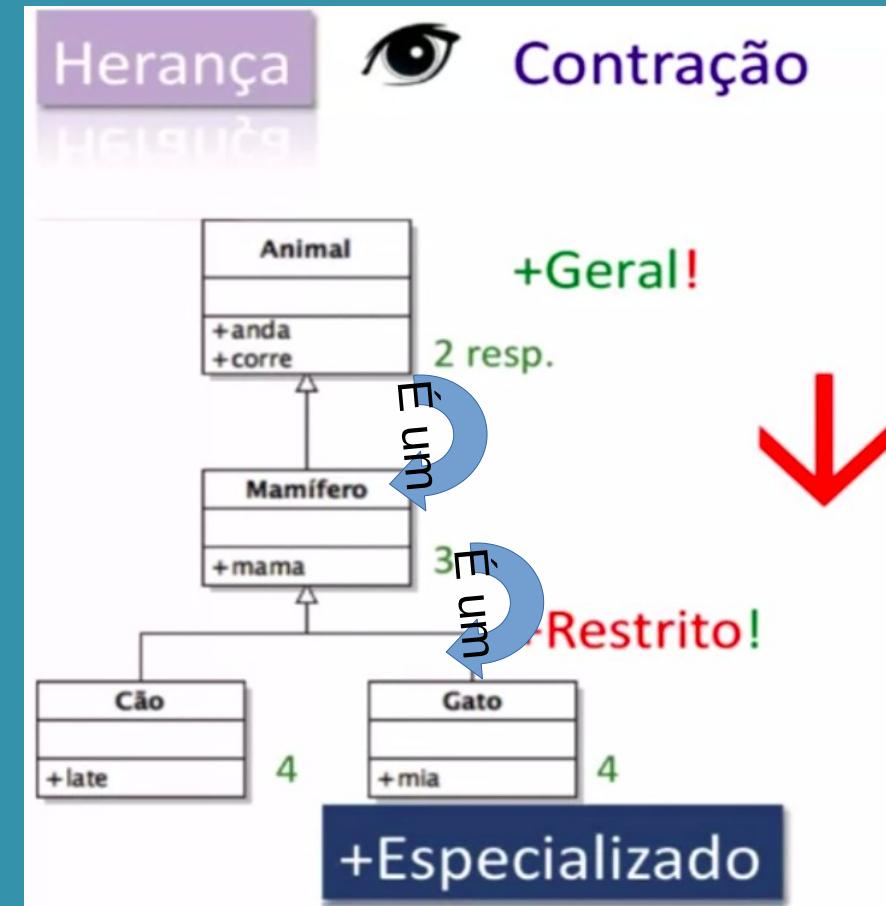


# Herança: Especialização e Generalização

## Generalização ↑



## Especialização ↓



# Modificadores de acesso

## Restringir o acesso a atributos e métodos

Acesso só por membros da própria classe

*private*

Qualquer classe pode acessar os membros

*public*

Cuidado!

Somente subclasses e classes têm acesso

*protected*

Acessível para classes do mesmo pacote

*default*

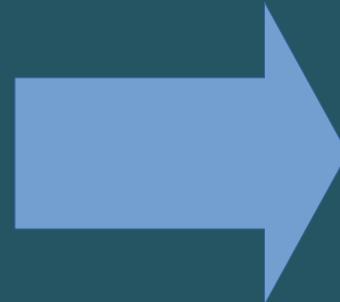
# Sobreposição de Métodos

```
public class Empregado{  
    public double liquido(){  
        return salario * 0.85;  
    }  
}
```



*precisa modificar o comportamento*

```
public class Gerente  
    extends Empregado{  
    //precisa adicionar  
    //o bônus depois do imposto  
}
```



```
public class Gerente  
    extends Empregado{
```

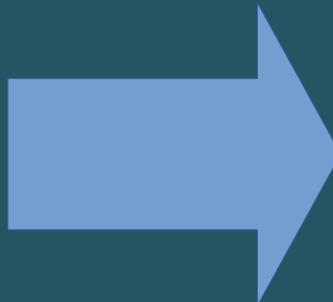
```
    public double liquido(){  
        return salario*0.85+bonus;  
    }
```



*Uma classe pode sobrescrever métodos da superclasse*

# Sobreposição de Métodos

```
public class Gerente  
    extends Empregado{  
  
    public double liquido(){  
        return salario*0.85+bonus;  
    }  
  
    
```



```
public class Gerente  
    extends Empregado{  
  
    public double liquido(){  
        return super.liquido()+bonus;  
    }  
  
    
```

*Você pode usar super para invocar o método da superclasse!*

# Proibindo modificadores nas classes

```
public final class Empregado{  
    public double liquido(){  
        return salario * 0.85;  
    }  
}
```



*Uma classe com final não pode ser estendida.*

```
public class Empregado{  
    public final double liquido(){  
        return salario * 0.85;  
    }  
}
```



*Um método com final não pode ser sobreescrito.*

# final em variáveis

```
final Empregado e = new Empregado();
```



*Não pode trocar a  
referência.*



*Pode modificar o  
objeto.*

*Diferente entre variáveis primitivas e de objeto!*

# CLASSES ABSTRATAS

Carro

!=

Veículo

?



Muito abstrato

*public abstract class Veiculo{*

...

}

# CLASSES ABSTRATAS

```
public abstract class Veiculo{  
    ...  
}
```

Uma classe abstrata só pode ser instanciada através de uma subclasse.

Classes abstratas definem um contrato e funcionalidades comuns a um grupo de classes.

```
public class Carro {  
    extends Veiculo{  
        ...  
    }
```

*Pode ser utilizada para ser estendida por outras classes!*

 `Veiculo v =  
new Veiculo();`

*Não pode ser instanciada diretamente!*

# CADEIA DE CONSTRutores

Superclasse

Subclasse

↑  
estende

Como inicializar  
na subclasse o  
que é da  
superclasse?



*Elas possuem seus próprios  
construtores para cuidar disso.*

Superclasse

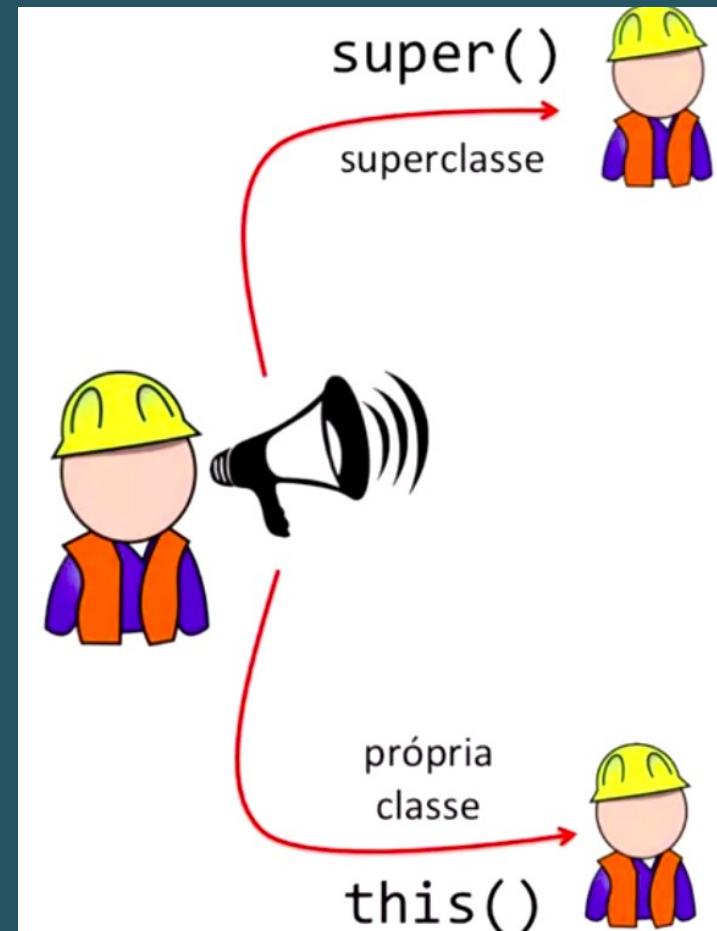
Atributo A

Atributo B

Atributo C

# CADEIA DE CONSTRutores

Em Java, um construtor sempre chama outro como a sua principal ação!



public class MinhaClasse{  
 private int var;  
 public MinhaClasse(int i){  
 var = i;  
 }  
}

como se fosse assim

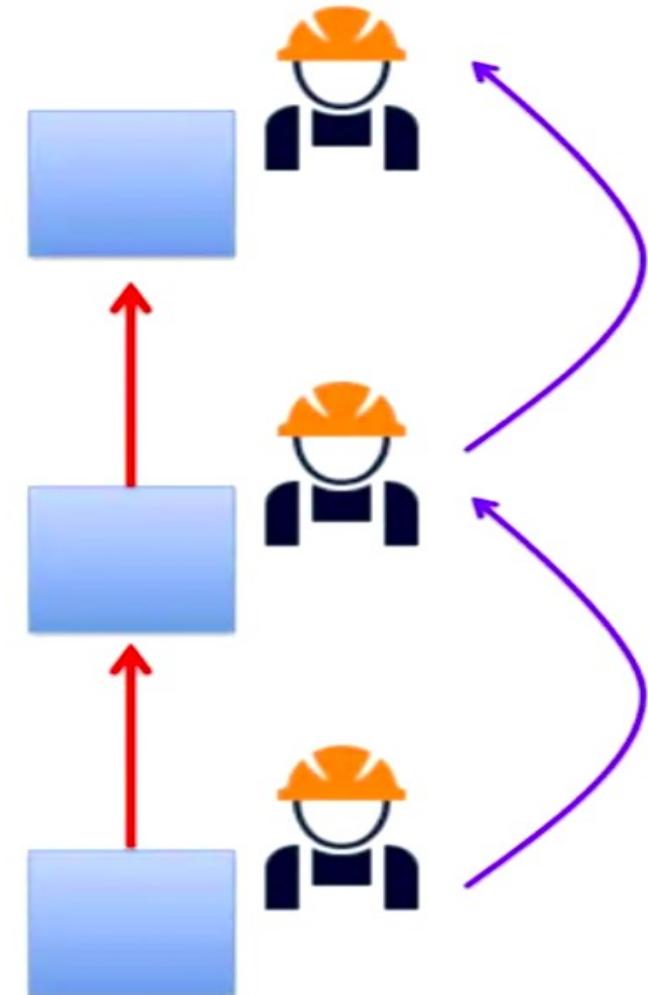
```
public class MinhaClasse{  
    private int var;  
    public MinhaClasse(int i){  
        super();  
        var = i;  
    }  
}
```

# CADEIA DE CONSTRutores

```
public class SuperC{  
    public SuperC(int i){}  
}  
  
public class SubC  
    extends SuperC{  
    public SubC(){  
        super(0);  
    }  
}
```



## Cadeia de Construtores



# ENCAPSULAMENTO

*Ninguém precisa saber como um celular funciona por dentro pra conseguir fazer ligações e usar os aplicativos.*

É necessário somente conhecer como interagir com a interface do celular e seus aplicativos.



**Identificar as classes que podem realizar suas funções de forma independentes (de forma encapsulada).**

get

```
public String getNome(){  
    return nome;  
}
```

set

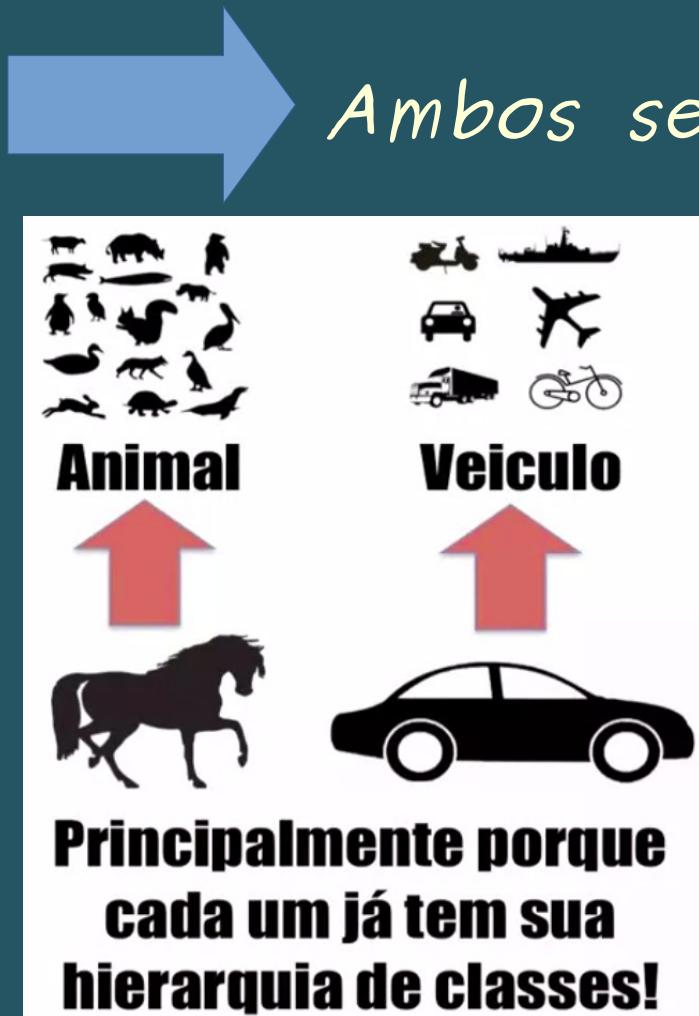
```
public String setNome(String  
nome){  
    this.nome = nome;  
}
```

Para um atributo somente-leitura, basta não adicionar o método “set”.

Um método “set” pode possuir uma lógica para apenas aceitar valores válidos.

O valor retornado pelo método “get” não precisa ser exatamente o que está na variável.

# INTERFACES

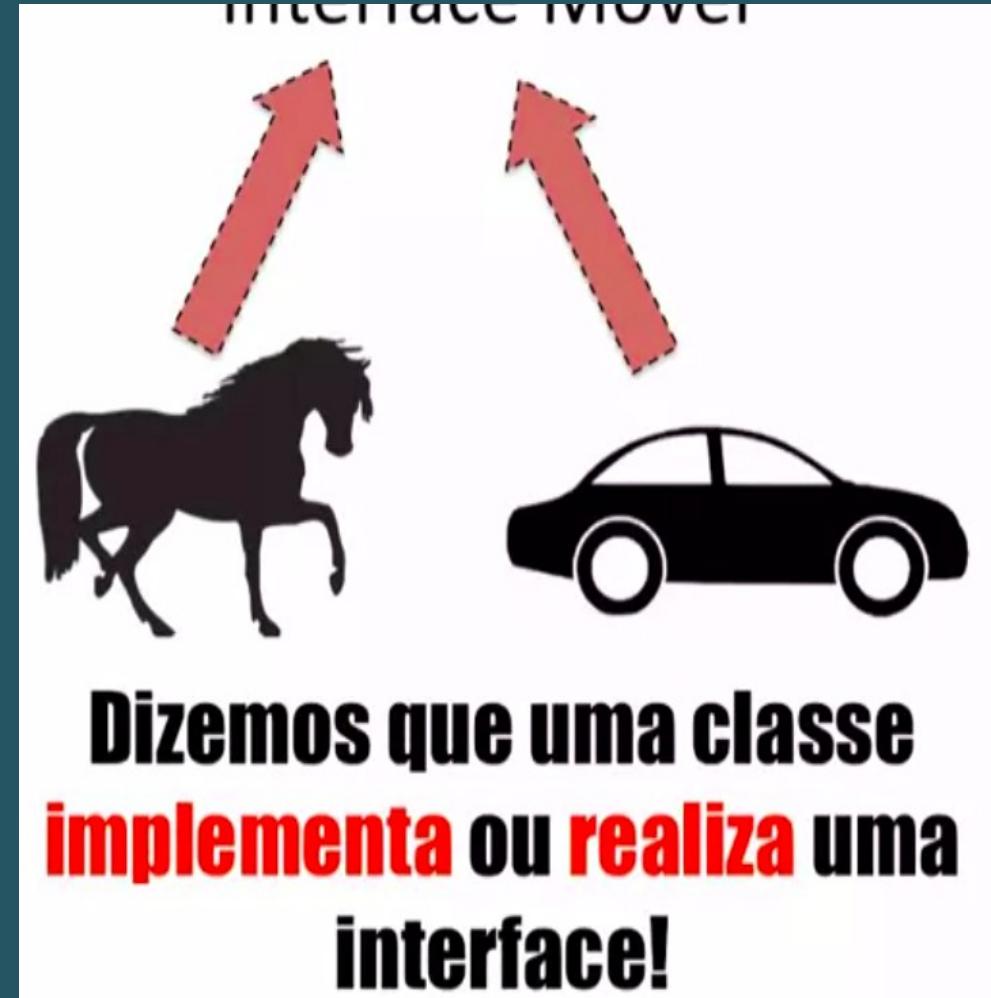


O que essas classes têm em comum é o comportamento

# INTERFACES

As interfaces podem ser utilizadas quando queremos abstrair um comportamento!

Define os métodos que uma classe precisar ter.



Dizemos que uma classe **implementa** ou **realiza** uma interface!

# INTERFACES

Interface Movel



```
public class Cavalo
    implements Movel{
    //precisa implementar todos
    //os métodos da interface
}
```

Interface Movel



```
public abstract class Veiculo
    implements Movel{
    //não precisa implementar todos
    //os métodos da interface,
    //mas subclasses concretas sim!
}
```

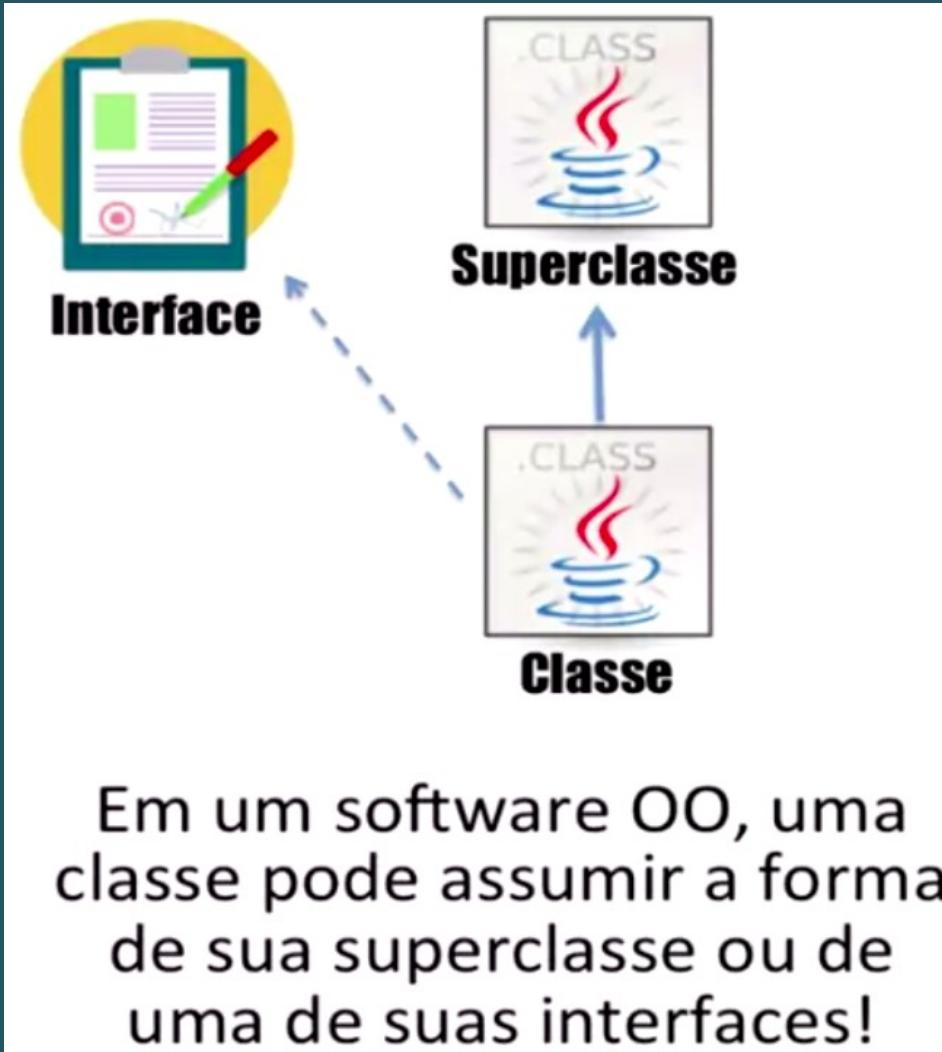
**Uma classe pode implementar uma ou mais interfaces!**

```
public class SmartTV
    implements Televisor,
    PlayerVideo{
    //deve implementar o contrato
    //de todas as suas interfaces
}
```



# POLIMORFISMO

SOUL CODE



```
Cavalo c = new Cavalo();  
Animal a = c;
```

**Um objeto do tipo Cavalo pode ser atribuído para uma variável do tipo Animal, sua superclasse!**

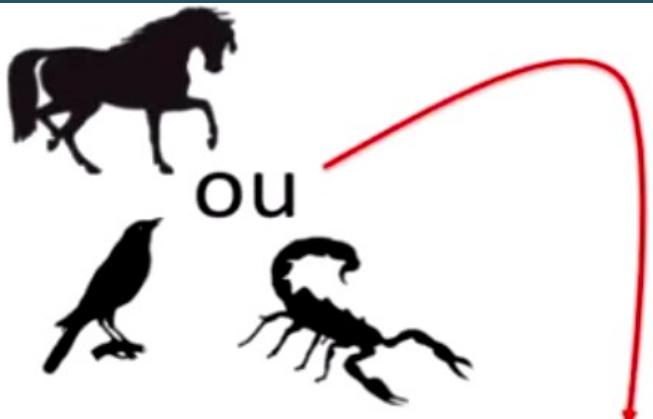
```
Cavalo c = new Cavalo();  
Movel m = c;
```

**Um objeto do tipo Cavalo pode ser atribuído para uma variável do tipo Movel, uma de suas interfaces.**

# POLIMORFISMO

O poder da orientação a objetos está no polimorfismo!

SOUL CODE



```
String classificar(Animal a)
```

**Um objeto pode ser sempre passado em parâmetros do tipo de sua superclasse.**

**O mesmo código pode ter diferentes comportamentos dependendo do objeto que está utilizando!**

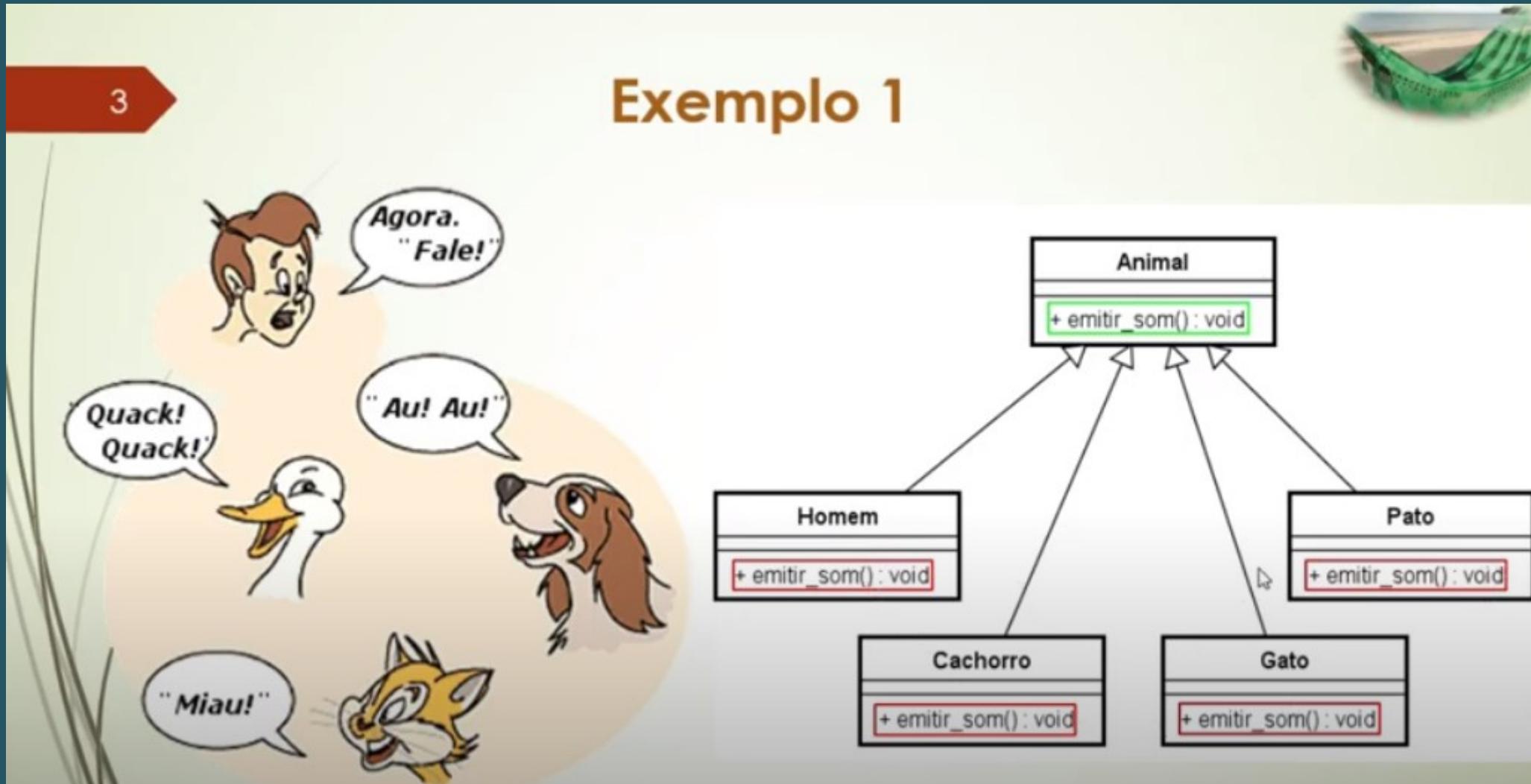


```
void addCorrida(Movel m)
```

**Um objeto pode ser sempre passado em parâmetros do tipo de uma de suas interfaces.**

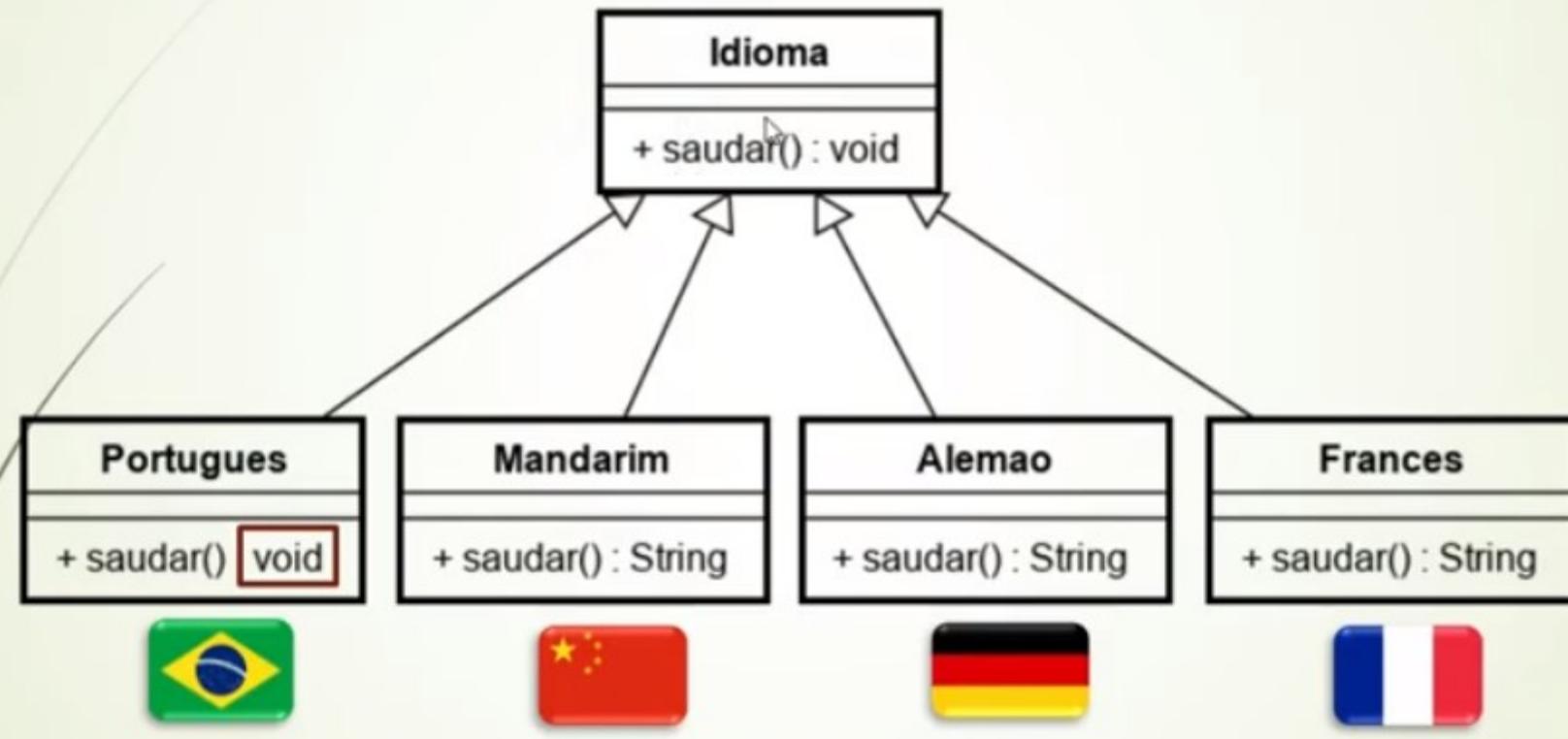
3

## Exemplo 1



4

## Exemplo 2



Fazer um programa para ler os dados de N contribuintes (N fornecido pelo usuário), os quais podem ser pessoa física ou pessoa jurídica, e depois mostrar o valor do imposto pago por cada um, bem como o total de imposto arrecadado.

Os dados de pessoa física são: nome, renda anual e gastos com saúde. Os dados de pessoa jurídica são nome, renda anual e número de funcionários. As regras para cálculo de imposto são as seguintes:

Pessoa física: pessoas cuja renda foi abaixo de 20000.00 pagam 15% de imposto. Pessoas com renda de 20000.00 em diante pagam 25% de imposto. Se a pessoa teve gastos com saúde, 50% destes gastos são abatidos no imposto. Exemplo: uma pessoa cuja renda foi 50000.00 e teve 2000.00 em gastos com saúde, o imposto fica:  $(50000 * 25\%) - (2000 * 50\%) = 11500.00$

Pessoa jurídica: pessoas jurídicas pagam 16% de imposto. Porém, se a empresa possuir mais de 10 funcionários, ela paga 14% de imposto. Exemplo: uma empresa cuja renda foi 400000.00 e possui 25 funcionários, o imposto fica:  $400000 * 14\% = 56000.00$

# EXCEÇÕES

SOUL CODE

*São classe especiais que representam erros e podem ser lançadas para quem invocou um método.*

```
public class MinhaException  
extends Exception{  
    //...  
}
```



**Para ser uma exceção, a classe precisa ser subclasse de Exception.**

# Tratamento de Exceção em Java

Programação Orientada a Objetos

SOUL CODE

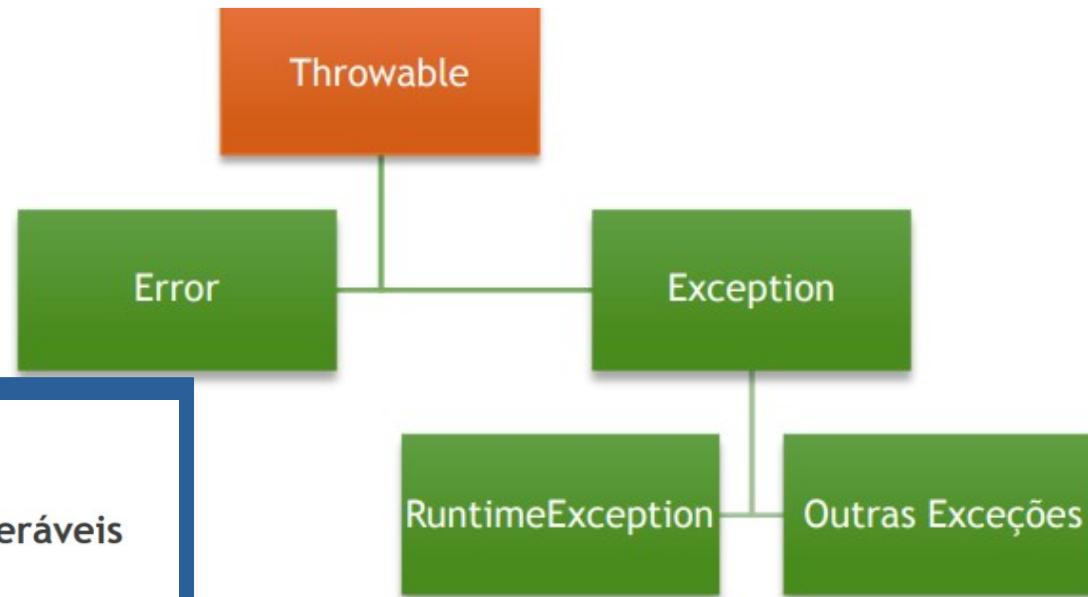
## O que são exceções?

- ▶ Uma **exceção** é uma condição anormal que altera ou interrompe o fluxo de execução.
  - ▶ Podem ser causadas por diversas condições:
    - ▶ Erros sérios de hardware;
    - ▶ Erros simples de programação;
    - ▶ Erros de divisão por zero;
    - ▶ Valores fora de faixa
    - ▶ Valores de variáveis
    - ▶ Erro na procura/abertura de um arquivo (entrada/saída)
    - ▶ Falha na memória
- ▶ Se não for tratada, o programa pode parar
  - ▶ O uso correto de exceções torna o programa mais robusto e confiável
  - ▶ Uso exagerado polui o código e torna o programa mais lento

# Tipos de exceção

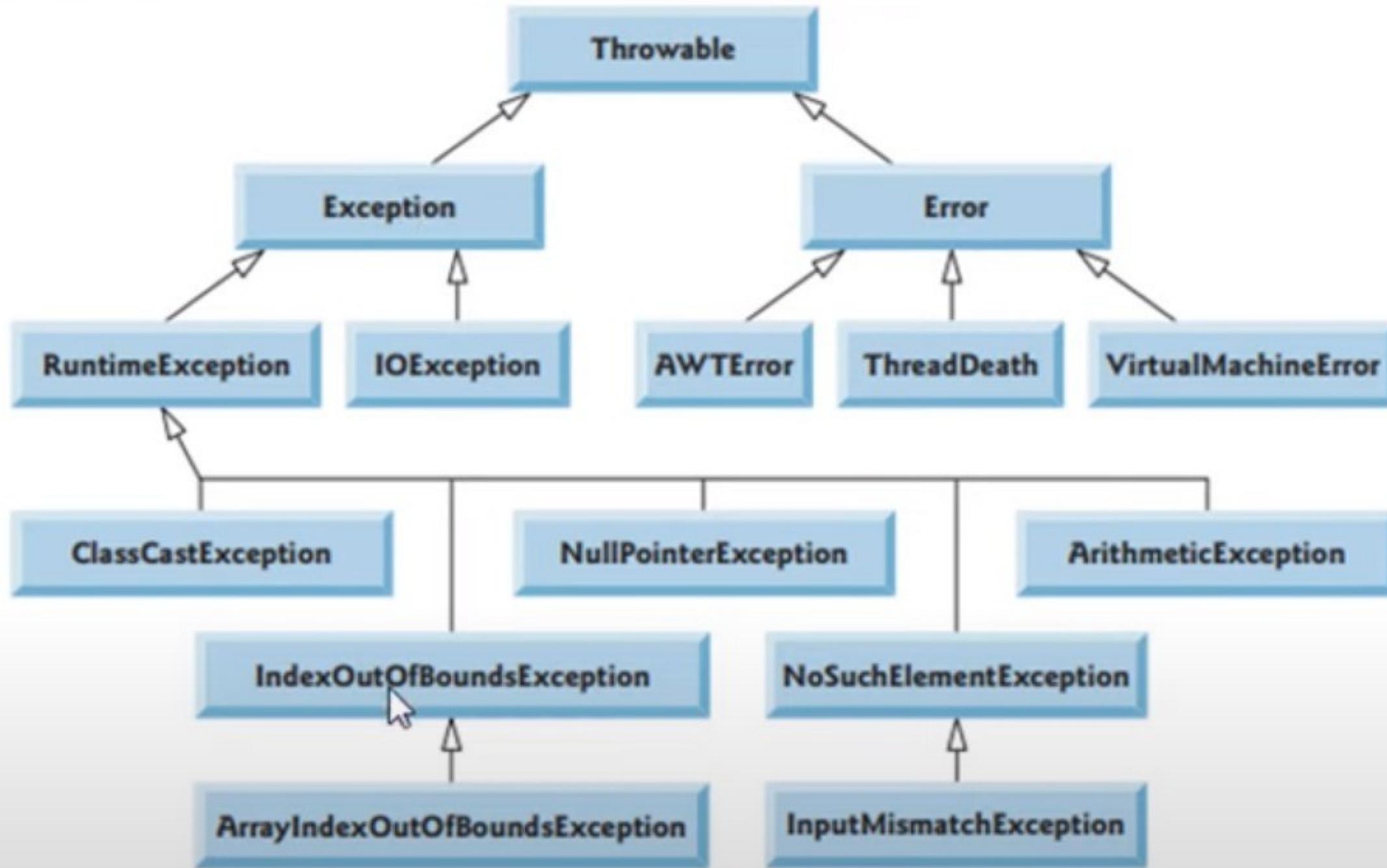
- ▶ Exceções, como (quase) tudo em Java, são objetos;

A classe Throwable oferece alguns métodos que podem verificar os erros reproduzidos, quando gerados para dentro das classes.



## Erros

- ▶ Os erros são tipicamente irrecuperáveis e apresentam condições sérias.



## java.lang.Throwable

- ▶ Ancestral de todas as classes que recebem tratamento do mecanismo de exceções;
- ▶ Principais métodos:
  - ▶ **void printStackTrace()**: lista a seqüência de métodos chamados até o ponto onde a exceção foi lançada;
  - ▶ **String getMessage()**: contém uma mensagem indicadora da exceção;
  - ▶ O método **toString()**: retorna uma descrição breve da exceção.

## java.lang.Exception

- ▶ Exceções que podem ser lançadas pelos métodos da API Java ou pelo seu programa;
- ▶ Devem ser tratadas;
- ▶ Em geral, representam situações inesperadas, porém contornáveis;
- ▶ O programador tem contato com esta classe e suas subclasses.

# Tipos de Exceções

- ▶ Exceções e erros se enquadram em **três categorias:**
  - ▶ Exceções verificadas;
  - ▶ Exceções não verificadas e;
  - ▶ Erros.

## Exceções Verificadas

- ▶ São **verificadas pelo compilador em tempo de compilação**;
- ▶ Os métodos que lançam uma exceção devem indicar isso na declaração do método através da cláusula **throws**.
- ▶ Todas as exceções verificadas devem ser capturadas explicitamente com um bloco **try – catch**;

## Exceções Não Verificadas

- ▶ Não verificadas em tempo de compilação;
- ▶ Ocorrem durante a execução por causa de erro do programador (divisão por zero, índice fora de faixa e uso de referências null);
- ▶ Não exigem tratamento;
- ▶ As exceções não verificadas incluem as exceções do tipo **RunTimeException** e todos os seus subtipos.
- ▶ A regra “Se for um RunTimeException, a culpa é sua” funciona muito bem.

# java.lang.RuntimeException

- ▶ Tipo especial de exceção;
- ▶ Não necessitam ser lançadas explicitamente pelo programa;
- ▶ Seu tratamento não é obrigatório; Ex.:
  - ▶ **NullPointerException**,
  - ▶ **IndexOutOfBoundsException**, etc.

## Palavras-chave

- Em Java, o código para tratamento de erros é separado de forma limpa do código que gera erros.
  - Diz-se que o código que gera a exceção “lança” (“**throw**”) uma exceção,
  - enquanto que o código que trata a exceção “captura” (“**catch**”) a exceção.

### A palavra-chave throw

- Para lançar uma exceção, use a palavra **throw**. Qualquer exceção verificada/não verificada, ou erro, pode ser lançado.

```
if (arquivo == null)  
    throw new FileNotFoundException();
```

## A palavra-chave throws

- ▶ O lugar onde você anuncia qual método pode lançar uma exceção é o cabeçalho do método;
- ▶ O cabeçalho muda para refletir as exceções verificadas que o método pode lançar.

```
public FileInputStream(String name) throws FileNotFoundException {  
    //...  
}
```

## Testando e capturando exceções

- ▶ Um bloco que tenta (try) chamar um método (ou conjunto de classes) que pode disparar uma exceção deve trata-la
  - ▶ Chamada normal de um método, mas que deve estar em um bloco try {...} catch {...}
- ▶ Uma exceção é um objeto que deve ser capturado (catch)
  - ▶ É nesse bloco que a exceção deve ser tratada
- ▶ Um trecho de código pode ser executado sempre
  - ▶ bloco finally

# Testando e capturando exceções

```
try
{
    //Executa código que pode disparar exceção
}
catch (Exception ex)
{
    //Trata exceção. ex é uma referência para o
    //objeto da classe Exception a ser tratado
}
finally
{
    //Esse bloco é sempre executado,
    //independente de ocorrer exceção.
}
```



# Exceções Frequentes

- ▶ ArithmeticException
- ▶ ClassNotFoundException
- ▶ DataFormatException
- ▶ FileNotFoundException
- ▶ IndexOutOfBoundsException
- ▶ NullPointerException
- ▶ NumberFormatException
- ▶ SQLException

```
Scanner scan = new Scanner(System.in);
System.out.println("Digite um número");
int numero = scan.nextInt();
```

*Sem tratamento da exceção*

```
Scanner scan = new Scanner(System.in);
System.out.println("Digite o valor de x: ");

try {
    int numero = scan.nextInt();
    System.out.println("valor digitado foi: " + numero);

} catch (InputMismatchException err) {
    System.out.println("Erro! O valor digitado não é válido. Tente novamente!");
}
```

*Com tratamento da exceção*

```
int[] numeros = {0,4,7,2,10};

Scanner scan = new Scanner(System.in);
System.out.println("Digite um número de 0 a 4: ");
int pos = scan.nextInt();

System.out.println("O valor que ocupa a posição " + pos + "é: " + numeros[pos] );
```

*Sem tratamento da exceção*

```
int[] numeros = {0,4,7,2,10};

Scanner scan = new Scanner(System.in);
System.out.println("Digite um número de 0 a 4: ";

try {
    int pos = scan.nextInt();
    System.out.println("O valor que ocupa a posição " + pos + "é: " + numeros[pos] );

} catch (IndexOutOfBoundsException err) {
    System.out.println("Erro! O valor digitado não é válido. Tente novamente!");
}
```

*Com tratamento da exceção*

## Testando e capturando exceções

- ▶ Se a exceção for disparada, o bloco **try** não será mais executado a partir do ponto onde a exceção ocorreu
- ▶ O fluxo de execução muda para a captura **catch**
- ▶ Pode haver mais de um bloco **catch**. A exceção é capturada pelo bloco que tratar a exceção correspondente.
- ▶ O bloco **finally** é opcional.

# Testando e capturando exceções

```
int x = 0;

try{
    int y = 100 / x;
    System.out.println ("Resultado: " + y);
}

catch (ArithmeticException e){
    System.out.println ("Operação inválida!");
    System.out.println("\n Detalhes do erro: "+ e.getMessage());
}
```

## A cláusula finally

- O código na cláusula finally executa se uma exceção foi ou não capturada.

```
int x = 0;
try{
    int y = 100 / x;
    System.out.println ("Resultado: " + y);
}
catch (ArithmeticException e){
    System.out.println ("Operação inválida!");
    System.out.println ("\n Detalhes do erro: "+ e.getMessage ());
}
finally{
    System.out.println ("Execução finalizada!");
}
```

# TRATAMENTO DE EXCEÇÕES

SOUL CODE

```
try{  
    //acessa o banco  
    //escreve arquivo  
}catch(IOException|SQLException e){  
    tratarErro(e);
```

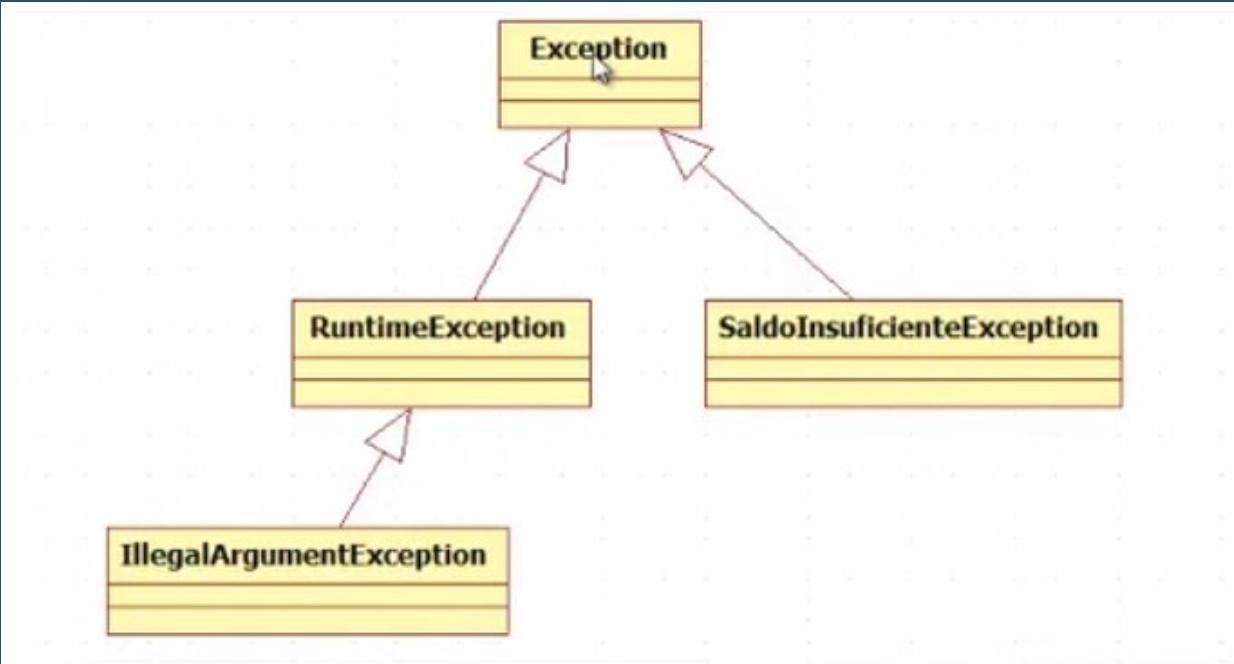
*É possível declarar um mesmo tratamento de erro para mais de um tipo de exceção.*

# TRATAMENTO DE EXCEÇÕES

SOUL CODE

*O bloco finally  
SEMPRE irá ser  
executado  
independente de  
acontecer uma  
exceção ou não!*

```
try{
    con=criaConexao();
    //acessa banco
}catch(SQLException e){
    trataErro(e);
}finally{
    con.close()
}
```



*Toda exceção checada extend diretamente a classe  
Exception → regras de negócio e precisa do try-catch*

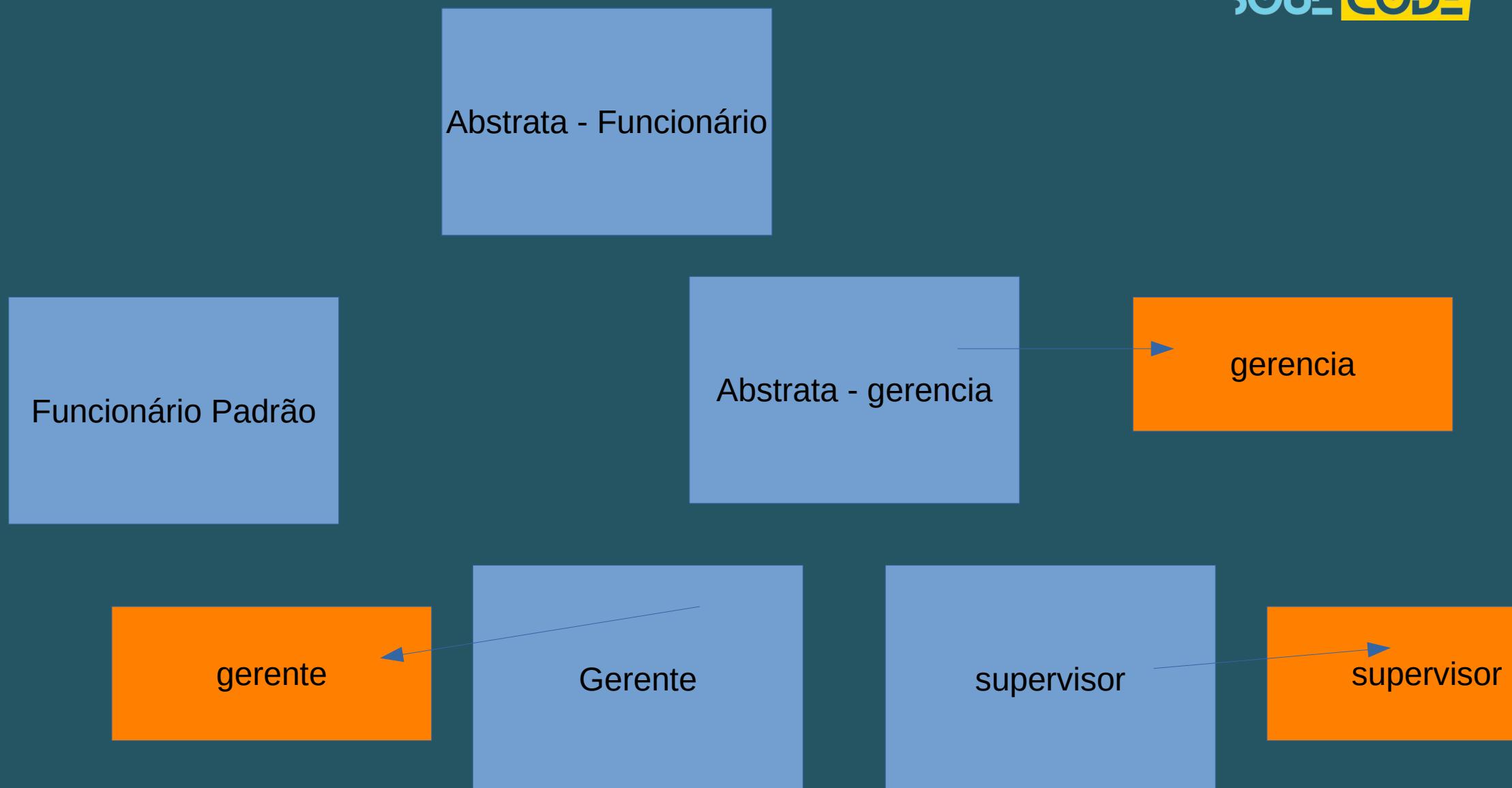
Fazer um programa para ler os dados de uma conta bancária e depois realizar um saque nesta conta bancária, mostrando o novo saldo. Um saque não pode ocorrer ou se não houver saldo na conta, ou se o valor do saque for superior ao limite de saque da conta. Implemente a conta bancária conforme projeto abaixo:

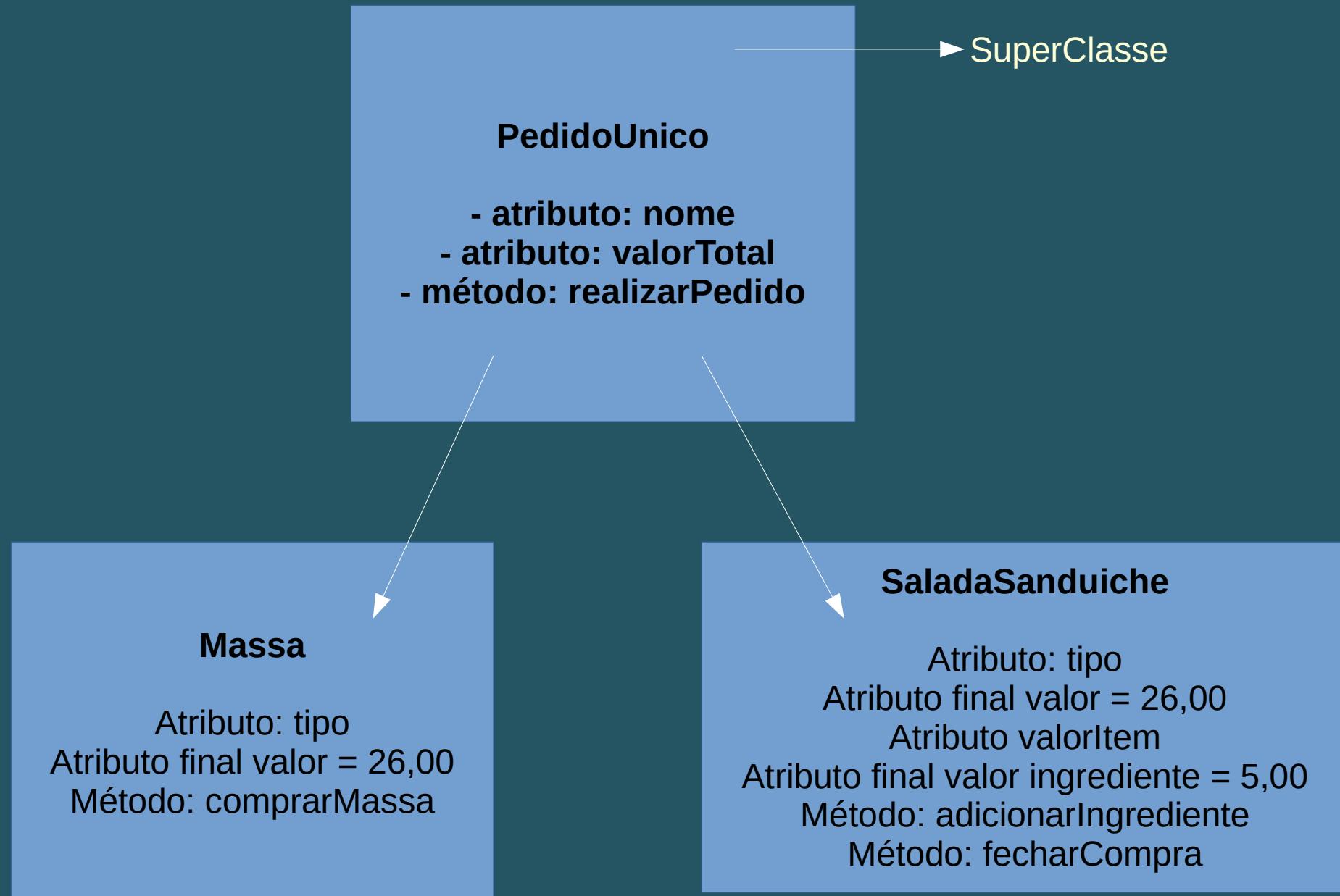
### CONTA

- numeroConta : Integer
- titular : String
- saldo : double
- limiteSaqui : double

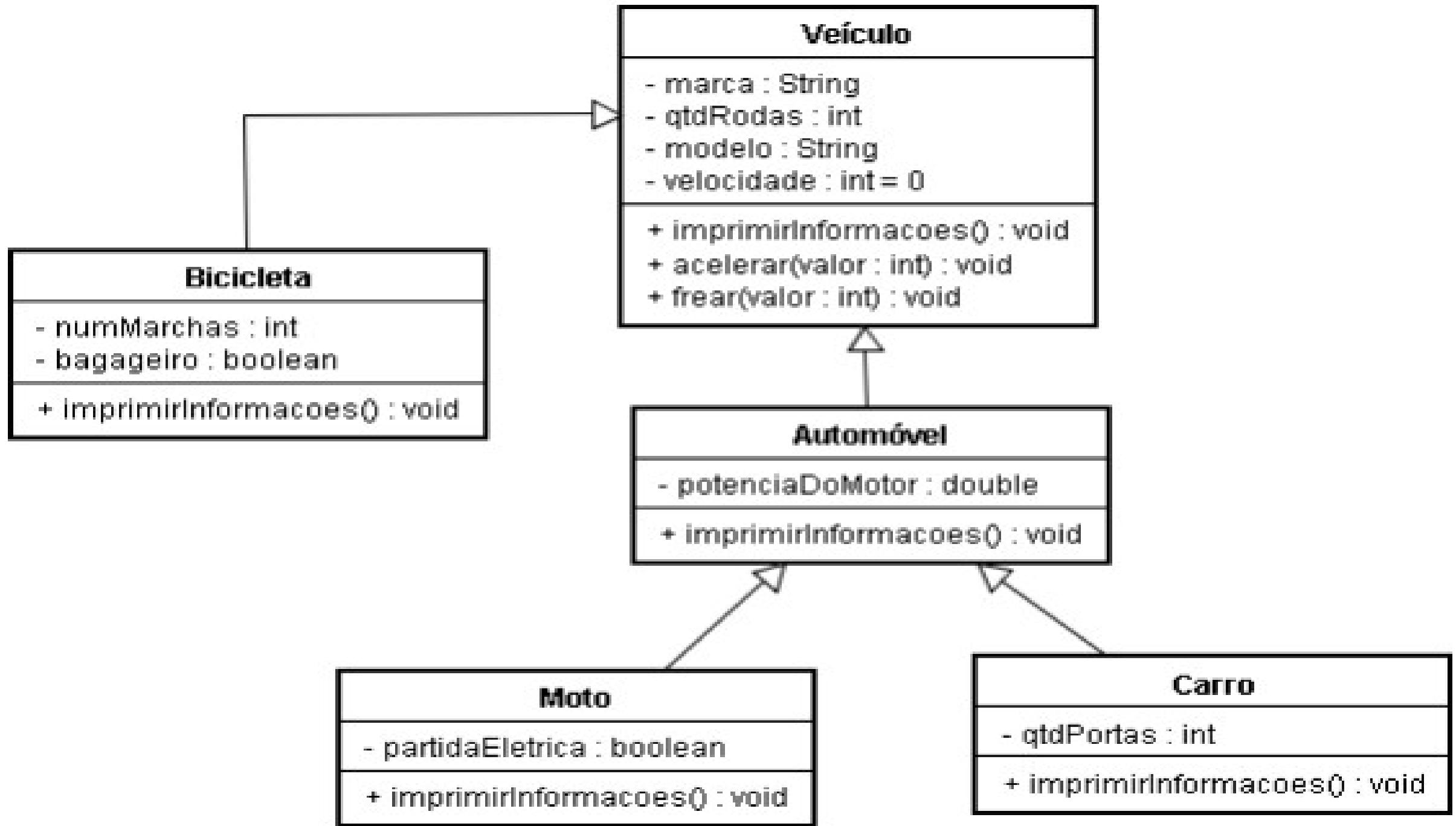
---

- +depositar
- +sacar









## Atividade – Implementação de um sistema para biblioteca

Nessa biblioteca é possível fazer a locação de livros, filmes e jogos de vídeo-game.

Regras: livros, filmes e jogos, deverão ser instanciados pelos atributos: código, nome e autor.

Regra para o empréstimo de livros: tem que haver mais de um no estoque. - Valor R\$5,00/dia

Regra para o empréstimo de filmes: o deve deve ter sido lançados no cinemas. - Valor R\$10,00/dia

Regra para o empréstimo de jogos: só emprestamos jogos para adultos. - Valor R\$ 7,00/dia

Métodos para todos os empréstimos:

- Emprestar
- calcular o valor empréstimo.

É necessário utilização dos conceitos de: herança, interface, polimorfismo e tratamento de excessões.

Os dados dos empréstimos devem ser lidos pelo prompt.

```
public void depositar(double valor){  
    System.out.println("-----" + this.getTitularConta() + "-----");  
    this.setSaldo(this.getSaldo() + valor);  
    if(this.getSaldo() >=0 ) {  
        //valor = valor - (this.getLimiteEspecial() - this.getSaldoLimite());  
        this.setSaldoLimite(this.getLimiteEspecial());  
    }  
    else{  
        this.setSaldoLimite(this.getSaldoLimite() + valor);  
    }  
}
```

## Conta Bancária

TitularConta  
NúmeroConta  
AgênciaConta  
Saldo Conta

-depositar  
-sacar

Conta Corrente

cartaoDebito

Conta Especial

chequeEspecial

Conta Poupança

rendimento

## Exemplo: Biblioteca - Trabalhando com Herança e Interfaces

O seu projeto deve ter:

- A classe Livro
- A interface LivroInterface
- A classe LivroFisico
- A classe Ebook
- Uma classe com o método main chamada Biblioteca

Classe Livro:

Atributos:

- nome;
- quantidade;

A classe Livro deve ser abstrata e implementar a Interface.

Regras

LivroFisico →

emprestar livro quando a quantidade suficiente.

- deve implementar os métodos da interface

Ebook →

Não precisa de quantidade suficiente.

- deve implementar os métodos da interface

## Exemplo: PagamentoImpostos - Trabalhando com Herança, Polimorfismo e Interfaces

O seu projeto deve ter:

- A classe Pessoa
- A interface PessoalInterface
- A classe PessoaFisica
- A classe PessoaJuridica
- Uma classe com o método main chamada Contribuicao

Classe Pessoa:

Atributos:

- nome;
- rendaAnual;

A classe Pessoa deve ser abstrata e implementar a Interface.

Regras

PessoaFisica →

- deve implementar o método calcularImposto
- Regras**

Se a renda anual < 20000 → pagar 15% de imposto.

Se maior → pagar 25% de imposto.

Se tiver gastosComSaude → diminuir o imposto em 50% dos gastos

PessoaJuridica →

- deve implementar o método calcularImposto
- Regras**

Se o numero de funcionários for mais que 10

→ pagar 14% sobre a renda

Se menor, pagar 20% sobre a renda

## Atividade: Figura - Trabalhando herança, polimorfismo e Interfaces

O seu projeto deve ter:

- A classe Figura
- A interface FiguralInterface
- A classe Retângulo
- A classe Circulo
- A classe Triangulo
- Uma classe com o método main chamada Matematica

**Classe Figura:**

**Atributos:**

- nome;
- cor;

A classe Figura deve ser abstrata e implementar a Interface.

A Interface deve o método : CalcularArea

Na classe Matematica deve ser instanciado Um objeto para cada subclasse e colocados em um lista, depois mostradas as áreas de cada objeto.

Retangulo →

- deve implementar o método calcularArea

**Regras**

Deve ter os atributos: lado1 e lado2

Circulo →

- deve implementar o método calcularArea

**Regras**

Deve ter o atributo: raio  
E um final pi

Triangulo →

- deve implementar o método calcularArea

**Regras**

Deve ter os atributos: base e altura