



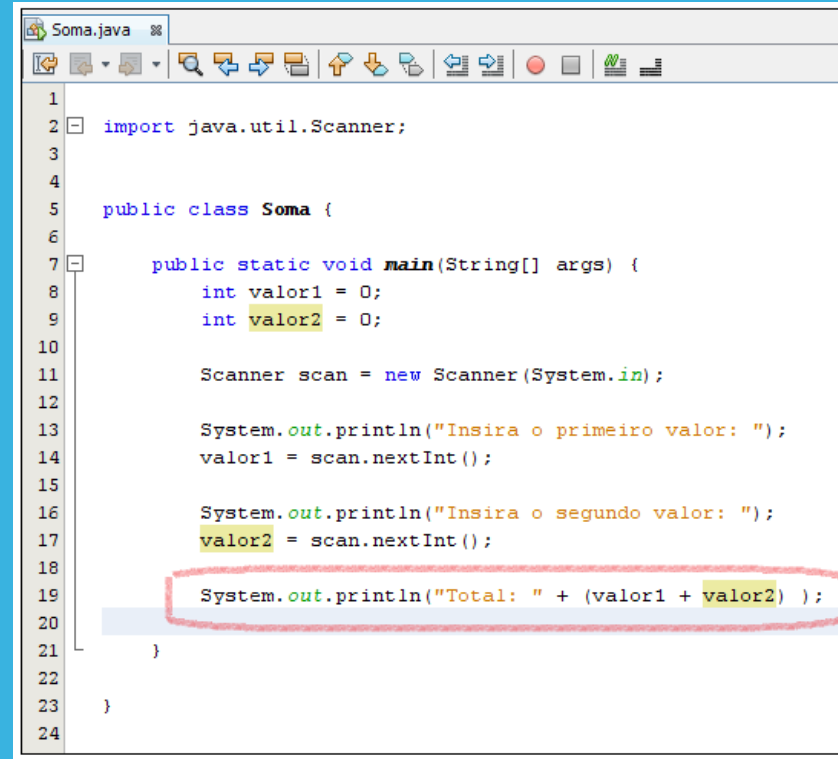
PROGRAMAÇÃO ORIENTADA A OBJETO - JAVA

PROFESSOR: DANILO FARIAS
SEMESTRE: 2020.1

June 6, 2022

ENTRADA DE DADOS PELO TECLADO

CLASSE MAIN, ATRIBUTOS, MÉTODOS E CONSTRUTORES



```
1
2 import java.util.Scanner;
3
4
5 public class Soma {
6
7     public static void main(String[] args) {
8         int valor1 = 0;
9         int valor2 = 0;
10
11         Scanner scan = new Scanner(System.in);
12
13         System.out.println("Insira o primeiro valor: ");
14         valor1 = scan.nextInt();
15
16         System.out.println("Insira o segundo valor: ");
17         valor2 = scan.nextInt();
18         System.out.println("Total: " + (valor1 + valor2));
19     }
20
21 }
22
23
24
```

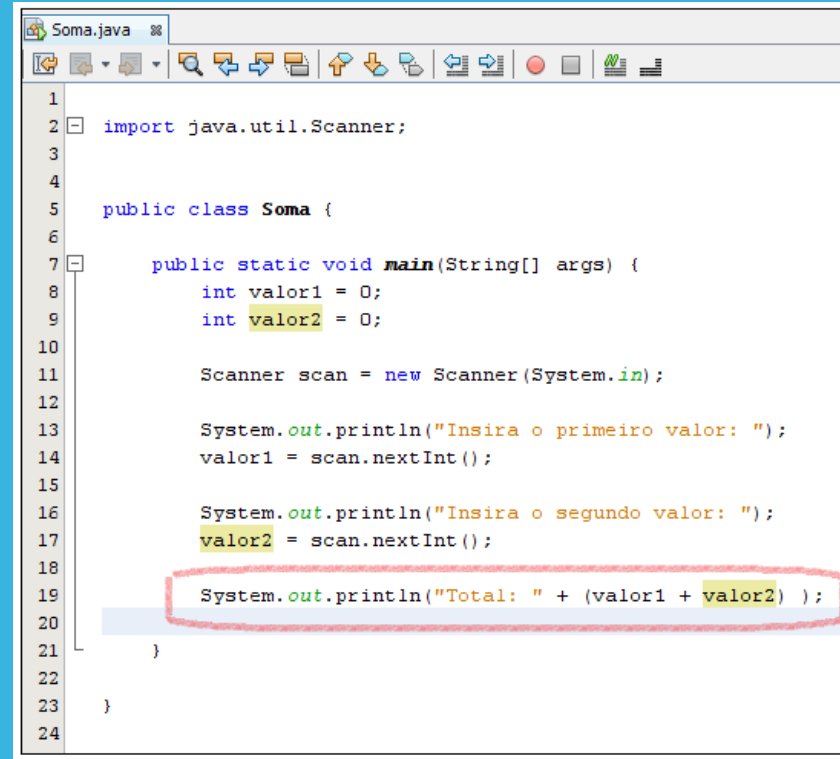
ENTRADA DE DADOS PELO TECLADO

- Ler dados a partir do teclado em Java não é algo tão simples quanto em outras linguagens de programação
- Existem três abordagens para leitura de dados a partir do:
 - Uso da classe *BufferedReader*;
 - Uso da classe **Scanner**;
 - Uso da classe *JOptionPane*;

Vamos ver apenas esse.

LENDO A PARTIR DA CLASSE SCANNER

CLASSE MAIN, ATRIBUTOS, MÉTODOS E CONSTRUTORES



```
1
2 import java.util.Scanner;
3
4
5 public class Soma {
6
7     public static void main(String[] args) {
8         int valor1 = 0;
9         int valor2 = 0;
10
11         Scanner scan = new Scanner(System.in);
12
13         System.out.println("Insira o primeiro valor: ");
14         valor1 = scan.nextInt();
15
16         System.out.println("Insira o segundo valor: ");
17         valor2 = scan.nextInt();
18
19         System.out.println("Total: " + (valor1 + valor2) );
20
21     }
22
23 }
24
```

SCANNER

- A partir da versão 5.0 da plataforma Java,
- Faz parte do pacote de classes utilitárias da plataforma Java (pacote `java.util`) e portanto para ser utilizada deve ser importada

SCANNER::EXEMPLO1

```
import java.util.Scanner;

public class LerTecladoScanner {

    public static void main(String[] args) {
        System.out.println("Digite algo e pressione (ENTER):");
        Scanner teclado = new Scanner(System.in);
        String dados = teclado.next();
        System.out.println("Você digitou: " + dados);
    }
}
```

SCANNER::PRÉ REQUISITOS

- É obrigatório que a classe `Scanner` seja importada.

```
import java.util.Scanner;
```

- Depois, um objeto do tipo ***Scanner*** é declarado. A definição deste objeto recebe como parâmetro a entrada padrão do sistema ***System.in***.

```
Scanner teclado = new Scanner(System.in);
```

SCANNER::MÉTODOS PARA LEITURA

- A classe Scanner oferece um conjunto de métodos para leitura de diferentes tipos de dados e que são apresentados na tabela abaixo:

Método	Finalidade
next()	Aguarda a entrada em formato String
nextInt()	Aguarda a entrada em formato Inteiro e utiliza o tipo de retorno int
nextByte()	Aguarda a entrada em formato Inteiro e utiliza o tipo de retorno byte
nextLong()	Aguarda a entrada em formato Inteiro Longo e utiliza o tipo de retorno long
nextFloat()	Aguarda a entrada em formato Fracionário e utiliza o tipo de retorno float
NextDouble()	Aguarda a entrada em formato Fracionário e utiliza o tipo de retorno double

SCANNER::EXEMPLO2

```
import java.util.Scanner;

public class LerTecladoScanner {

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

        System.out.println("Digite um número inteiro e pressione (ENTER):");
        int dados = teclado.nextInt();
        System.out.println("Você digitou: " + dados);
    }
}
```

ARRAYS EM JAVA

CLASSE MAIN, ATRIBUTOS, MÉTODOS E CONSTRUTORES

```
import java.util.*  
public class Array2{  
    private static final String colors[] = { "vermelho", "azul", "verde",  
        "amarelo", "preto", "vermelho", "azul", "rosa"};  
  
    public Array2(){  
        //copio meu array para uma lista - asList()  
        List<String> list = Arrays.asList(colors);  
  
        System.out.printf("ArrasList: %s\n", list);  
    }  
  
    public static void main(String args[]){  
        new Array2();  
    }  
}
```

June 6, 2022

INTRODUÇÃO

- Em praticamente toda linguagem de programação há necessidade de se guardar e tratar elementos como um único conjunto
- Em linguagens de programação esta idéia é geralmente tratada através do conceito de vetores e matrizes (também chamados de arrays).
- Arrays são tipos especiais em Java
- Definem estruturas que armazenam valores de um determinado tipo (primitivo ou por referência)
- Arrays são objetos, portanto são considerado tipos por referência

ARRAYS

- Declaração

`int[] idades;` (Mais usado e recomendado)

ou

`int idades[];`

- A variável `idades` é uma referência, pois um array é um objeto

- Criando um array de int de 10 posições

`idades = new int[10];`

- Declarando e criando um array de int de 10 posições

`int[] idades = new int[10];`

ARRAYS

- Atalhos de sintaxe para criar e inicializar um array

```
int[] idades= {5, 6, 8, 9, 10, 20, 25, 30, 40, 50};
```

ARRAYS



○ Array multidimensional

```
String[][] nomes = new String[2][3];
```

```
nomes[0][0] = "Mr.";
```

```
nomes[0][1] = "Mrs.";
```

```
nomes[0][2] = "Ms.";
```

```
nomes[1][0] = "Smith";
```

```
nomes[1][1] = "Jones";
```

```
nomes[1][2] = "David";
```

ou

```
String[][] nomes = {{ "Mr.", "Mrs.", "Ms." }, { "Smith", "Jones", "David" }};
```

○ Acessando

```
System.out.println(nomes[0][0] + nomes[1][0]); //Mr. Smith
```

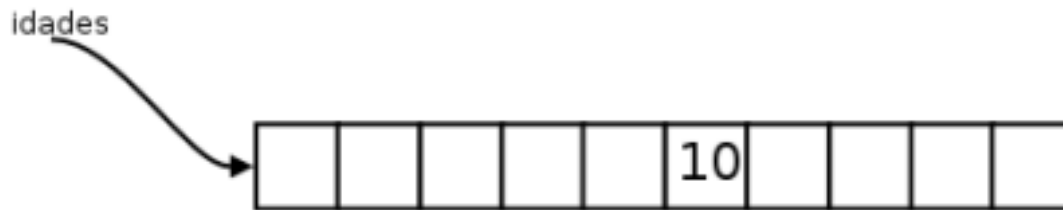
```
System.out.println(nomes[0][2] + nomes[1][1]); //Ms. Jones
```



ARRAYS

- Acessando as posições de um array

`idades[5] = 10`



ARRAYS

- Arrays de Referência

```
Conta[] minhasConta;  
minhasConta = new Conta[10];
```

- Quantas contas foram criadas aqui?

- Resposta: **Nenhuma**. Foram criados 10 espaços para guardar uma referência a uma Conta

ARRAYS

- Estado atual



ARRAYS

- Populando arrays

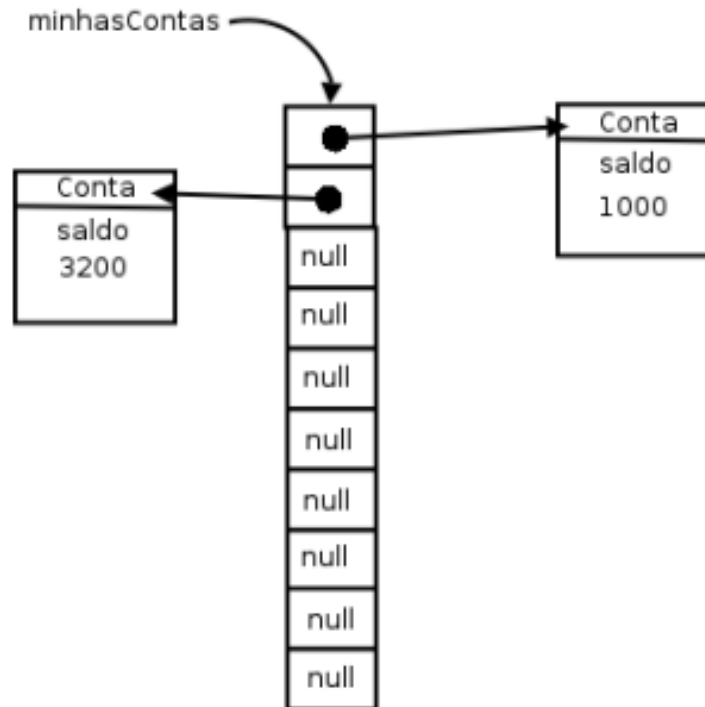
```
Conta contaNova = new Conta();  
contaNova.setSaldo(1000.00);  
minhasContas[0] = contaNova;
```

//Ou

```
minhasContas[1] = new Conta();  
minhasContas[1].setSaldo(3200.00);
```

ARRAYS

- Estado atual



ARRAYS

- Percorrendo um array

```
public static void main(String args[]) {  
    int[] idades = new int[10];  
    for (int i = 0; i < 10; i++) {  
        idades[i] = i * 10;  
    }  
    for (int i = 0; i < 10; i++) {  
        System.out.println(idades[i]);  
    }  
}
```

ARRAYS

○ O atributo **length**

- O atributo `length` retorna o tamanho do array
- Muito útil quando recebemos um array como retorno de algum método

```
void imprimeArray(int[] array) {  
    for (int i = 0; i < array.length; i++) {  
        System.out.println(array[i]);  
    }  
}
```

- Obs: Assim que um array é criado, ele não pode mudar de tamanho.

ARRAYS

- Se é feito acesso a um elemento indefinido de um array, é gerado uma exceção:
 - `IndexOutOfBoundsException`

```
String[] nomes = {"José", "João", "Pedro"};  
System.out.println(nomes[3]);
```

Gera um erro em tempo de execução

ARRAYS – EXERCÍCIO PRÁTICO

- Agora vamos praticar:

- No projeto do sistema bancário vamos criar a Classe Transação que terá os atributos:

- idTransacao :: int;
- tipo :: TipoTransacao * (Enum)
- data :: Date;
- valor :: double;

- Por fim criar um ArrayList de transações na Conta;
 - Crie o método de lista transações (Extrato)

```
enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }  
  
class Card1 {  
    Suit suit;  
  
}  
  
public class SuitTest1 {  
    public static void main(String args[]){  
        Card1 card=new Card1();  
        card.suit=Suit.CLUBS;  
    }  
}
```

ARRAYS – EXEMPLO DE ARRAY LIST

ArrayExample1.java

```
package com.mkyong;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class ArrayExample1 {

    public static void main(String[] args) {

        String[] str = {"A", "B", "C"};

        List<String> list = new ArrayList<>(Arrays.asList(str));

        list.add("D");

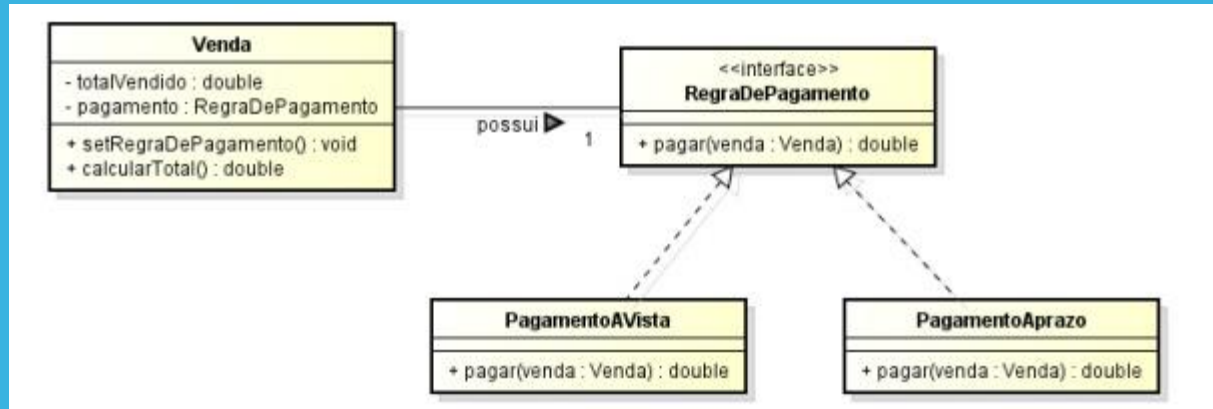
        list.forEach(x -> System.out.println(x));

    }
}
```

June 6, 2022

HERANÇA

HERANÇA SIMPLES E MULTINÍVEIS



HERANÇA

- É o mecanismo pelo qual pode-se definir uma nova classe de objetos a partir de uma classe já existente
- Esta nova classe poderá aproveitar o comportamento e possíveis atributos da classe estendida
- A classe sendo refinada é chamada de superclasse ou classe base, enquanto que a versão refinada da classe é chamada uma subclasse ou classe derivada

HERANÇA

○ Usar quando:

- Desejamos estender funcionalidades ou características a partir de um tipo de dado (classe) existente no sistema;
- Identificamos no sistema vários tipos de dados (classes) com características e funcionalidades comuns porém, cada um deles contendo também suas características e funcionalidades particulares.

HERANÇA

○ Exemplo: Acrescentando comportamentos

Conta
- numero : String - saldo : double
+ creditar(valor : double) : void + debitar(valor : double) : void + getSaldo() : double + getNumero() : String

```
class Conta{  
    public Conta(String numero) {}  
  
    public void creditar (double valor) {}  
    public void debitar (double valor) {}  
    public String getNumero() {}  
    public double getSaldo() {}  
}
```

Poupanca
- numero : String - saldo : double
+ creditar(valor : double) : void + debitar(valor : double) : void + getSaldo() : double + getNumero() : String + renderJuros(taxa : double) : void

```
class Poupanca{  
    public Poupanca(String numero) {}  
  
    public void creditar (double valor) {}  
    public void debitar (double valor) {}  
    public String getNumero() {}  
    public double getSaldo() {}  
    public void renderJuros(double taxa) {}  
}
```

HERANÇA

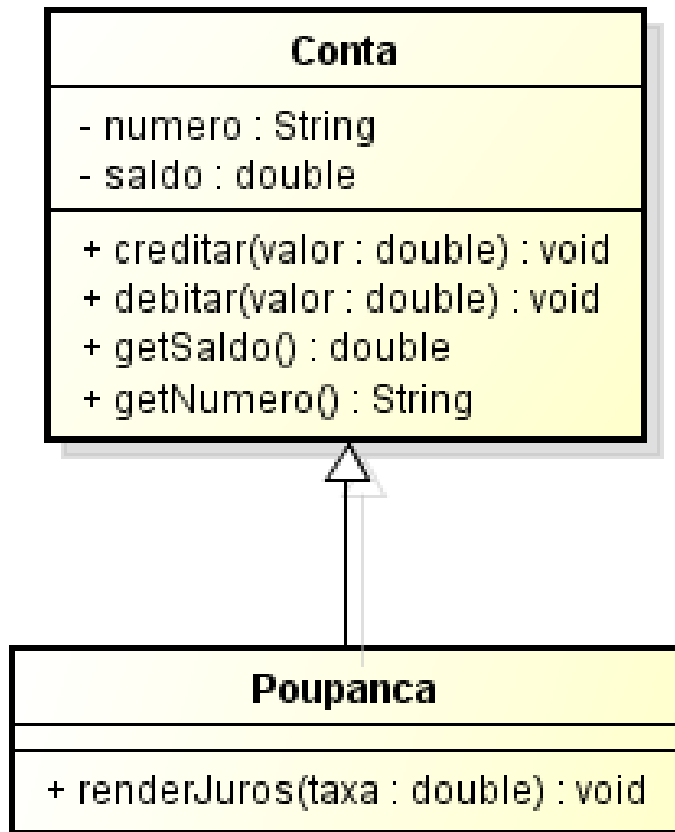
○ Problemas

- Duplicação desnecessária de código
- Uma poupança também é uma conta

HERANÇA

- Quando dizemos que:
 - Uma classe B herda de uma classe A ou
 - Uma classe B é subtipo de uma classe A ou
 - Uma classe B é subclasse de uma classe A
- Significa dizer que todos os atributos e métodos que foram definidos em A, dependendo do modificador de acesso, também fazem parte de B.
- A palavra reservada utilizada para expressar o conceito de herança em Java é extends.

HERANÇA



```
class Conta{
    public Conta(String numero) {}

    public void creditar (double valor) {}
    public void debitar (double valor) {}
    public String getNumero() {}
    public double getSaldo() {}
}
```

```
class Poupanca extends Conta {

    public Poupanca (String numero) {
        super(numero);
    }

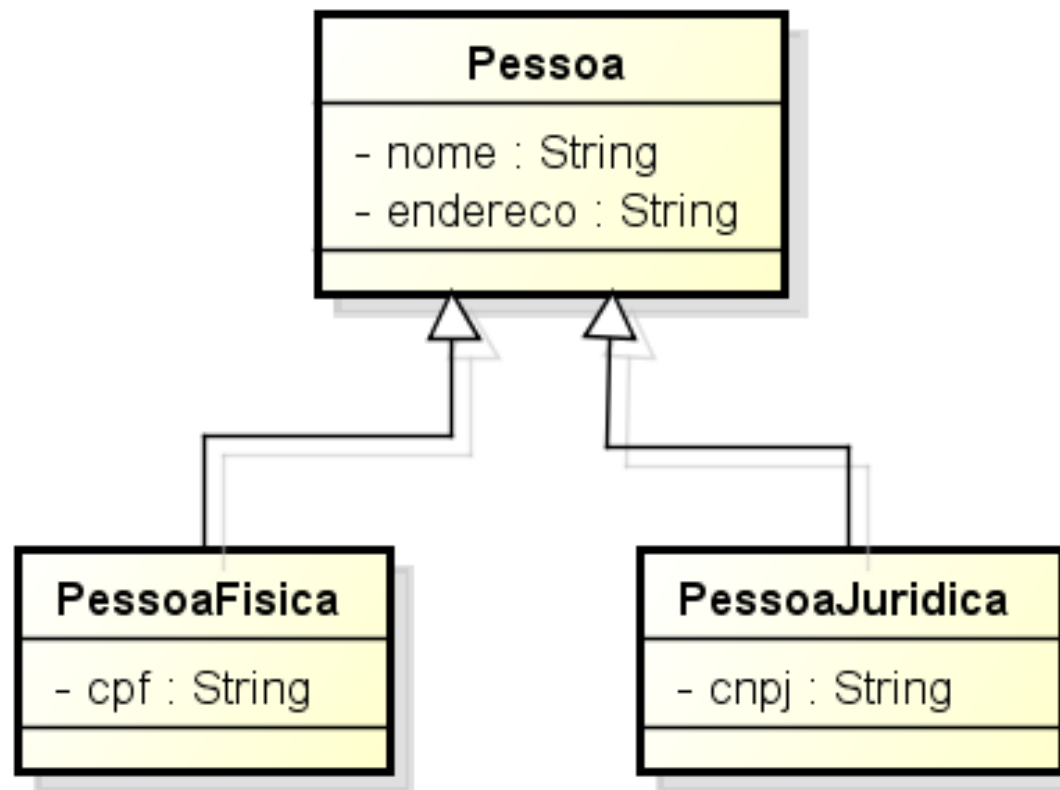
    public void renderJuros(double taxa) {
        double saldoAtual = getSaldo();
        creditar(saldoAtual * taxa);
    }
}
```

HERANÇA

- Encadeamento de construtores
 - No construtor da **subclasse** devemos fazer uma chamada ao construtor da **superclasse**.
 - Em seguida, devemos fazer a **inicialização** dos **atributos** da própria **subclasse**.
 - Uso da palavra reservada ***super***.
 - Referenciar explicitamente definições (construtores, métodos, atributos) que foram implementadas na superclasse.

HERANÇA

○ Exemplo



HERANÇA



```
public class Pessoa {  
  
    private String nome;  
    private String endereco;  
  
    public Pessoa(String n, String end) {  
        this.setNome(n);  
        this.setEndereco(end);  
    }  
}
```

```
public class PessoaFisica extends Pessoa {  
  
    private String cpf;  
  
    public PessoaFisica(String n, String end, String cpf) {  
        super(n, end);  
        this.cpf = cpf;  
    }  
}
```

```
public class PessoaJuridica extends Pessoa{  
  
    private String cnpj;  
  
    public PessoaJuridica(String n, String end, String cnpj) {  
        super(n, end);  
        this.setCnpj(cnpj);  
    }  
}
```

HERANÇA

- Exemplo de uso:

```
public class Teste {  
  
    public static void main(String[] args) {  
        PessoaFisica pessoaFisica = new PessoaFisica("João", "Rua da Hora", "123.123.123-00");  
        imprime(pessoaFisica);  
    }  
  
    public static void imprime(PessoaFisica pessoaFisica) {  
        System.out.println("Nome: "+pessoaFisica.getNome());  
        System.out.println("Endereço: "+pessoaFisica.getEndereco());  
        System.out.println("CPF: "+pessoaFisica.getCpf());  
    }  
}
```

HERANÇA

- Instanciando nossas classes. Problemas?

```
...  
Poupanca p;  
p = new Poupanca("19.222-0");  
p.creditar(100.00);  
p.debitar(50.00);  
p.renderJuros(0.0123);  
...
```

HERANÇA

- Instanciando nossas classes. Problemas?

```
...  
Poupanca p;  
p = new Poupanca("19.222-0");  
p.creditar(100.00);  
p.debitar(50.00);  
p.renderJuros(0.0123);  
...
```

```
...  
Conta c;  
c = new Poupanca("19.222-0");  
c.creditar(100.00);  
c.debitar(50.00);  
c.renderJuros(0.0123);  
...
```

HERANÇA

- Instanciando nossas classes. Problemas?

```
...  
Poupanca p;  
p = new Poupanca("19.222-0");  
p.creditar(100.00);  
p.debitar(50.00);  
p.renderJuros(0.0123);  
...
```

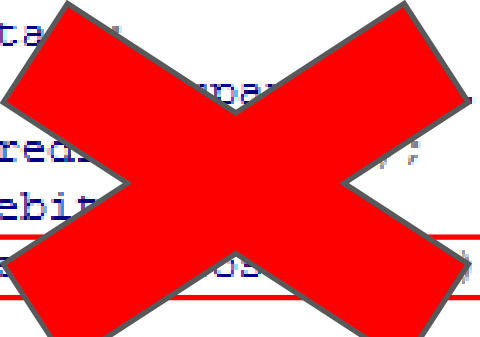
```
...  
Conta c;  
c = new Poupanca("19.222-0");  
c.creditar(100.00);  
c.debitar(50.00);  
c.renderJuros(0.0123);  
...
```

HERANÇA

- Instanciando nossas classes. Problemas?

```
...  
Poupanca p;  
p = new Poupanca("19.222-0");  
p.creditar(100.00);  
p.debitar(50.00);  
p.renderJuros(0.0123);  
...
```

```
...  
Conta c;  
c = new Conta("19.222-0");  
c.creditar(100.00);  
c.debitar(50.00);  
c.renderJuros(0.0123);  
...
```



HERANÇA

○ Resolvendo nosso problema

- O que acontece é que o compilador toma as suas decisões com base apenas no tipo declarado para a variável.
- Contudo, Java dispõe de mecanismos que permitem contornar este problema:
 - Conversão de tipos: **cast**
 - O operador **instanceof**
 - Permite determinar qual o tipo de um objeto que está sendo referenciado por uma variável.

HERANÇA

○ Resolvendo nosso problema

- Usando Cast para informar ao compilador que temos certeza que a Conta informada é uma Poupança

Cast

```
...  
Conta c;  
c = new Poupanca("19.222-0");  
c.creditar(100.00);  
c.debitar(50.00);  
((Poupanca) c).renderJuros(0.0123);  
...
```

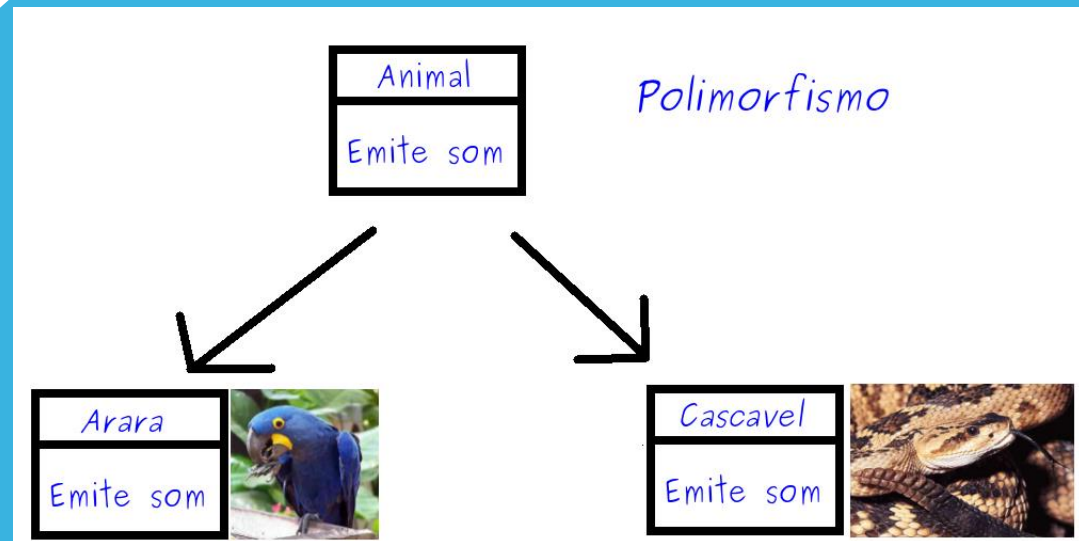
HERANÇA

- E quando não temos certeza que o objeto instanciado é uma conta ou uma poupança?
- Operador **instanceof**

```
...  
Conta c = procura("22.233-6");  
  
if (c instanceof Poupanca) {  
    ((Poupanca) c). renderJuros(0.0123);  
} else {  
    System.out.print("Poupança inexistente!");  
}  
...
```

POLIMORFISMO

SOBRESCRITA E SOBRECARGA



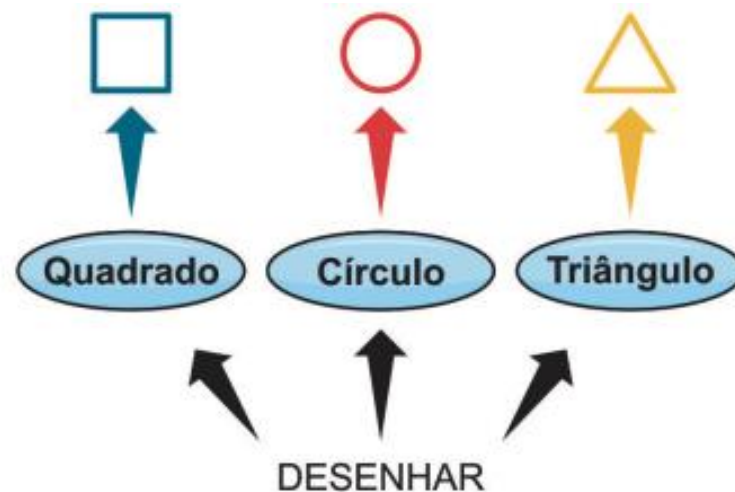
June 6, 2022

POLIMORFISMO

- É originário do grego, e quer dizer “muitas formas”
- Significa que um mesmo tipo de objeto, sob certas condições, pode realizar ações diferentes ao receber uma mesma mensagem

POLIMORFISMO

- Permite o envio de uma mesma mensagem a objetos distintos, onde cada objeto responde da maneira mais apropriada

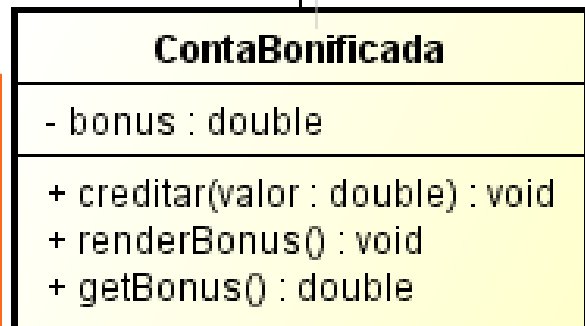
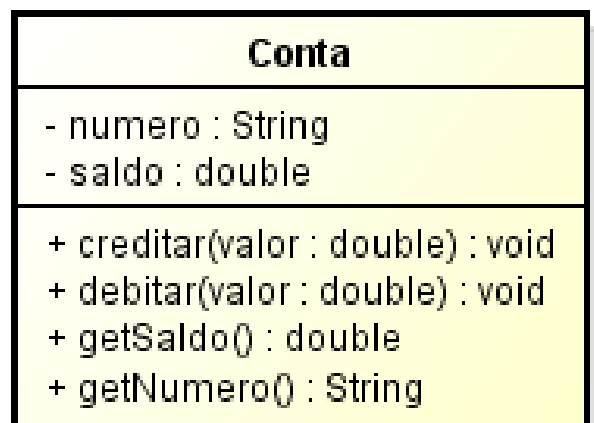


POLIMORFISMO

- Todas as classes Java herdam da classe `java.lang.Object`

POLIMORFISMO

○ Sobreposição (Sobrescrita ou *Override*)



```
class ContaBonificada extends Conta {  
    private double bonus;  
  
    public ContaBonificada(String numero){  
        super(numero);  
    }  
  
    public void creditar(double valor) {  
        this.bonus = this.bonus + (valor * 0.01);  
        super.creditar(valor);  
    }  
  
    public void renderBonus() {  
        super.creditar(bonus);  
        this.bonus = 0;  
    }  
  
    public double getBonus() {  
        return this.bonus;  
    }  
}
```

POLIMORFISMO



○ Sobreposição (Sobrescrita ou *Override*)

- Todas as classes Java herdam da classe **java.lang.Object**
- O método `toString`

- Método da classe **java.lang.Object**
- As classes podem reescrever esse método para mostrar uma mensagem, uma `String`, que o represente

```
Conta c = new Conta();  
System.out.println(c.toString());
```

- O método `toString` do **Object** retorna o nome da classe @ um número de identidade:

```
Conta@34f5d74a
```

- Mas isso não é interessante



POLIMORFISMO

○ Sobreposição (Sobrescrita ou *Override*)

- O método toString
 - Vamos reescrever o método toString

```
class Conta {  
    private double saldo;  
    // outros atributos...  
  
    public Conta(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public String toString() {  
        return "Uma conta com valor: " + this.saldo;  
    }  
}
```

- E agora teremos:

```
Conta c = new Conta(100);  
System.out.println(c.toString()); //imprime: Uma conta com valor: 100.
```

POLIMORFISMO

- Sobreposição (Sobrescrita ou *Override*)
 - O método toString
 - O método toString é chamado pelo método **println**

```
Conta c = new Conta(100);  
System.out.println(c); //imprime: Uma conta com valor: 100.
```

POLIMORFISMO

- Sobreposição (Sobrescrita ou *Override*)
 - O método equals
 - Método da classe `java.lang.Object`
 - Quando comparamos duas variáveis referência no Java, o `==` verifica se as duas referem-se ao mesmo objeto:

```
Conta c1 = new Conta(100);
Conta c2 = new Conta(100);
if (c1 != c2) {
    System.out.println("objetos referenciados são diferentes!");
}
```
 - Podemos sobrescrever o método equals para compara atributos do objeto
 - O **equals** recebe um **Object** como argumento e deve verificar se ele mesmo é igual ao Object recebido para retornar um boolean.
 - Se não reescrever esse método, o comportamento herdado é fazer um `==` com o objeto recebido como argumento.

POLIMORFISMO

- Sobreposição (Sobrescrita ou *Override*)
 - O método equals

```
public class Conta {  
    private String numero;  
    private float saldo;  
  
    public Conta(String numero, float saldo) {  
        this.numero = numero;  
        this.saldo = saldo;  
    }  
  
    public boolean equals(Object obj) {  
        Conta other = (Conta) obj;  
        if (!this.numero.equals(other.numero)) {  
            return false;  
        }  
        return true;  
    }  
}
```

POLIMORFISMO

- Sobreposição (Sobrescrita ou *Override*)

- O método equals

```
public class Teste {  
  
    public static void main(String[] args) {  
        Conta conta1 = new Conta("12.222-0",100);  
        Conta conta2 = new Conta("13.332-0",140);  
  
        if(conta1.equals(conta2)){  
            System.out.println("Contas iguais");  
        }else{  
            System.out.println("Contas diferentes");  
        }  
    }  
}
```

- Resultado: **Contas diferentes**

POLIMORFISMO

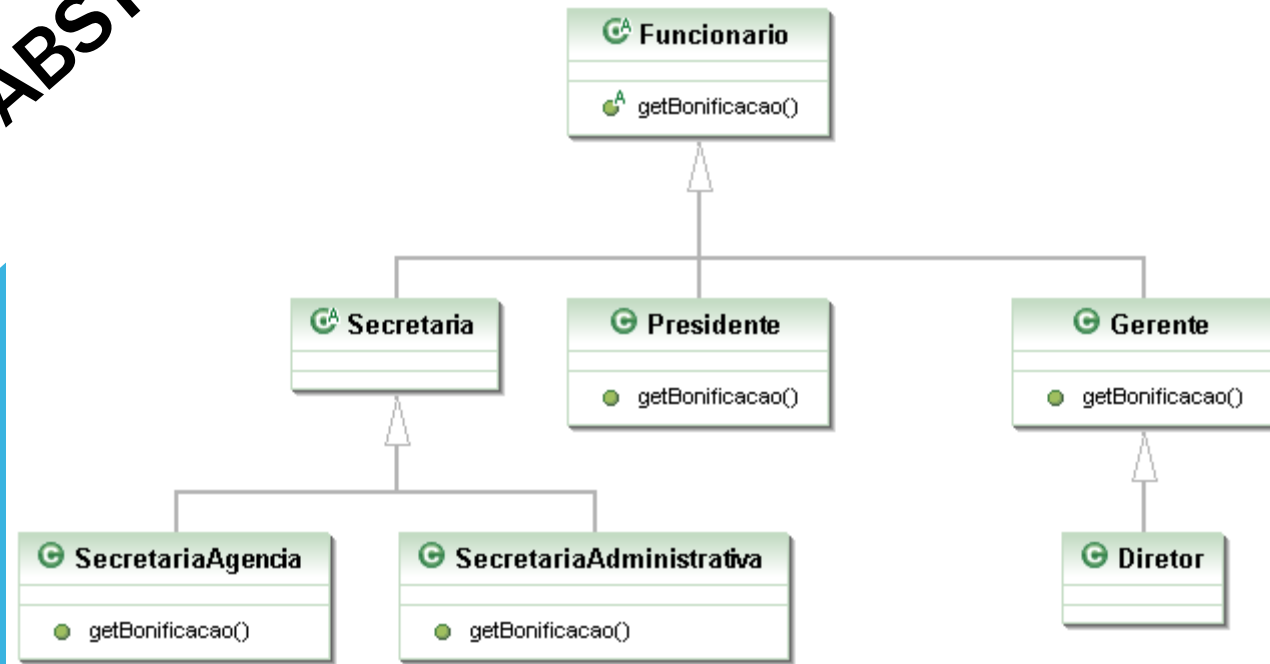
○ Sobrecarga ou *Overload*

- Permite uma classe ter mais de um método com o mesmo nome;
- Cada método é diferenciado pelos tipos e quantidades de parâmetros;

- Exemplo:

```
public class Retangulo {  
    public double area(double lado) {  
        return lado * lado;  
    }  
  
    public double area(double largura, double comprimento) {  
        return largura * comprimento;  
    }  
}
```

CLASSE ABSTRATA



CLASSE ABSTRATA

- Uma classe abstrata contem membros que serão reutilizados para a criação de um novo tipo de classe
- Não é possível instanciar uma classe abstrata
- Podem conter membros que são úteis em uma hierarquia de classes, mas são tão genéricas que acabam não tendo significado se isoladas destas classes

CLASSE ABSTRATA

- Simplificam o reuso de código
- Definem “contratos” a serem realizados por subclasses
- Tornam o polimorfismo mais claro
- Devem ser declaradas com a palavra-chave `abstract`
- Podem declarar métodos abstratos
 - Métodos sem implementação
 - Implementação fornecida na subclasse
- Podem declarar métodos concretos
 - Métodos com implementação

CLASSE ABSTRATA

○ Estrutura:

```
public abstract class NomeClasse {  
    atributo1;  
    atributo2;  
  
    ...  
    public void metodoConcreto() {  
        //Código  
    }  
  
    ...  
    public abstract void metodoAbstrato();  
}
```

CLASSE ABSTRATA

○ Exemplo:

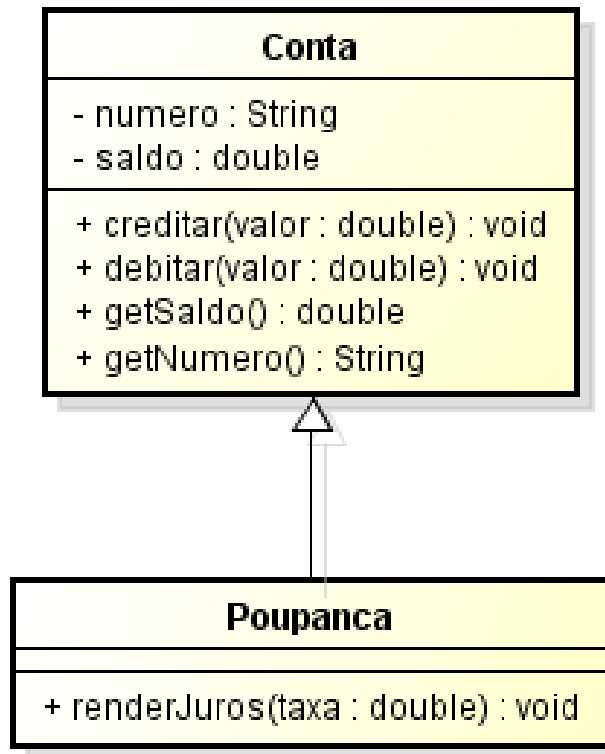
```
public abstract class Poligono {  
    private int numeroLados;  
    public Poligono(int numeroLados) {  
        super();  
        this.numeroLados = numeroLados;  
    }  
    public int getNumeroLados() {  
        return numeroLados;  
    }  
    public void setNumeroLados(int numeroLados) {  
        this.numeroLados = numeroLados;  
    }  
    public abstract int perimetro();  
}
```

CLASSE ABSTRATA

○ Exemplo:

```
public class Quadrado extends Poligono {  
    private int tamanhoLado;  
    public Quadrado(int tamanhoLado) {  
        super(4);  
        this.tamanhoLado = tamanhoLado;  
    }  
    public int getTamanhoLado() {  
        return tamanhoLado;  
    }  
    public void setTamanhoLado(int tamanhoLado) {  
        this.tamanhoLado = tamanhoLado;  
    }  
    public int perimetro() {  
        return this.getNumeroLados() *  
               this.getTamanhoLado();  
    }  
}
```

CLASSES CONTA E POUPANÇA



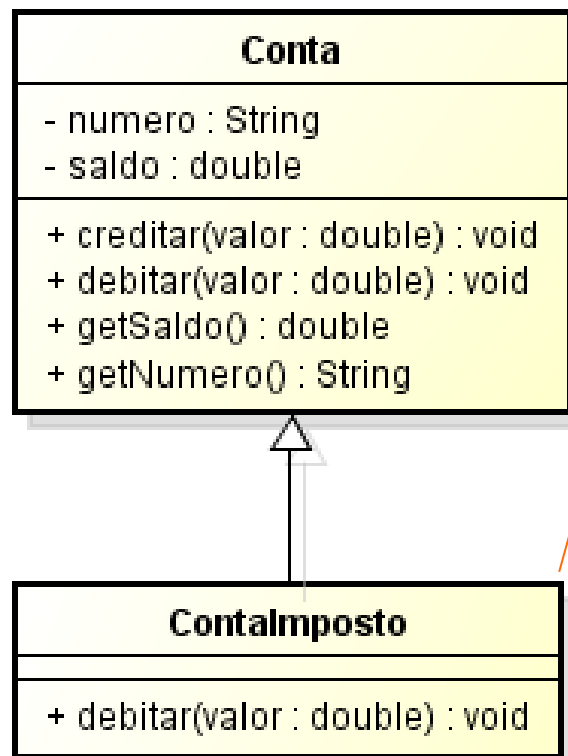
```
class Conta{
    public Conta(String numero) {}

    public void creditar (double valor) {}
    public void debitar (double valor) {}
    public String getNumero() {}
    public double getSaldo() {}
}
```

```
class Poupanca extends Conta {

    public Poupanca (String numero) {
        super(numero);
    }
    public void renderJuros(double taxa) {
        double saldoAtual = getSaldo();
        creditar(saldoAtual * taxa);
    }
}
```

CONTA IMPOSTO

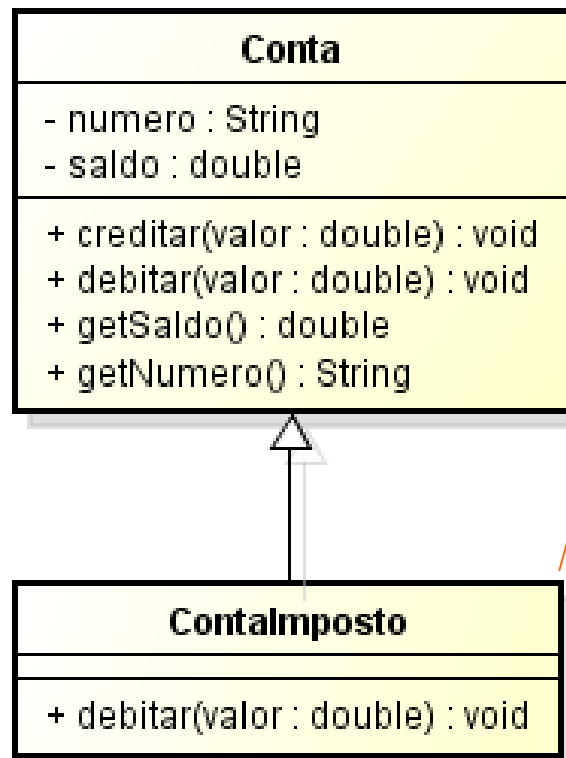


```
public class ContaImposto extends Conta {
    public static final double TAXA = 0.001;

    public ContaImposto (String numero) {
        super(numero);
    }

    public void debitar(double valor) {
        double importado = valor * TAXA;
        super.debitar(valor + importado);
    }
}
```

CONTA IMPOSTO

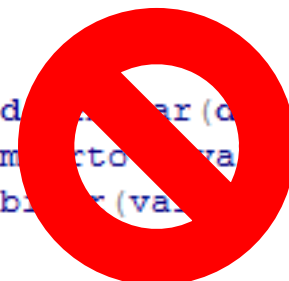


```

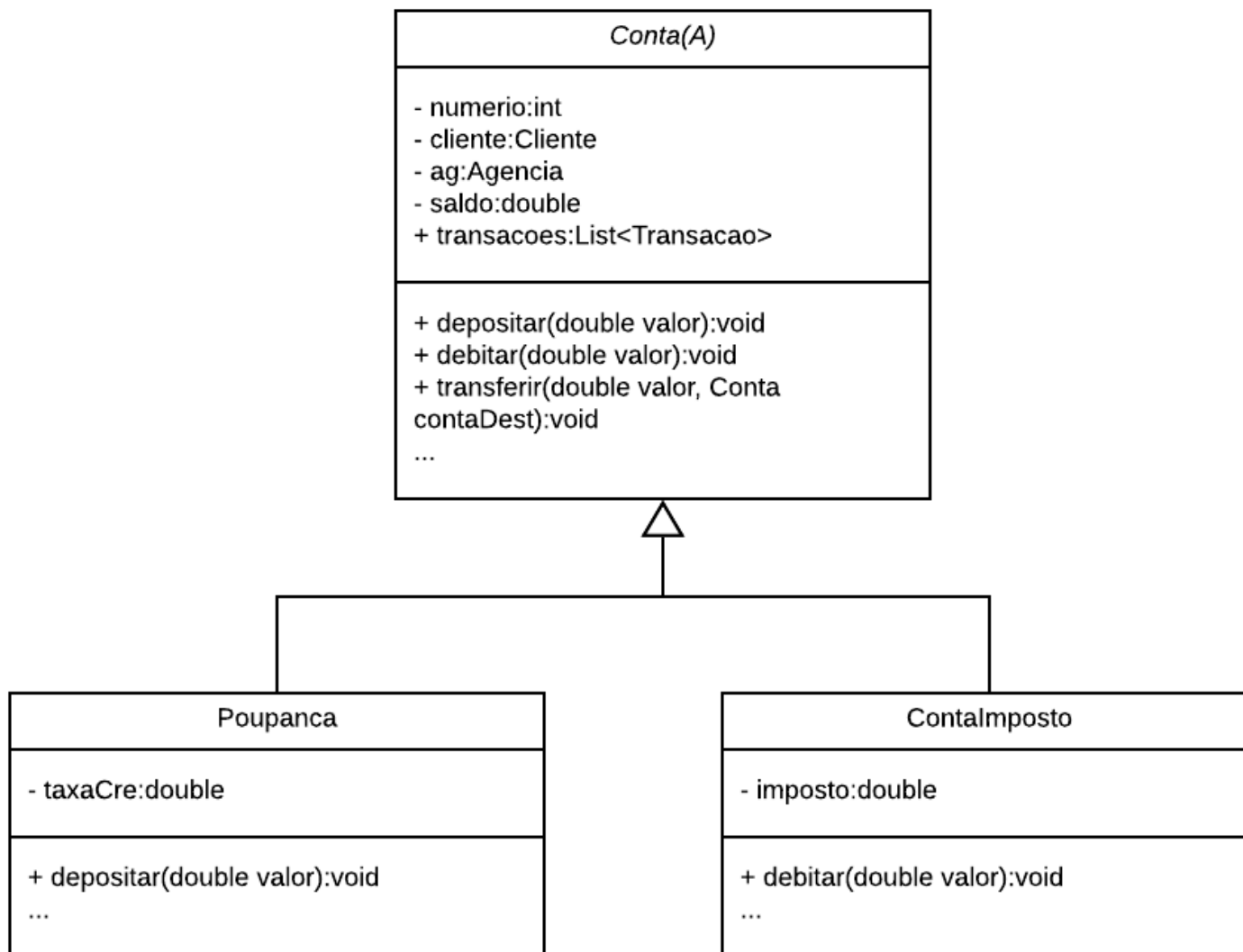
public class ContaImposto extends Conta {
    public static final double TAXA = 0.001;

    public ContaImposto (String numero) {
        super (numero);
    }

    public void debitar (double valor) {
        double imposto = valor * TAXA;
        super.debitar (valor + imposto);
    }
}
  
```



AGORA TEMOS



NOVA CLASSE – CONTA ABSTRATA

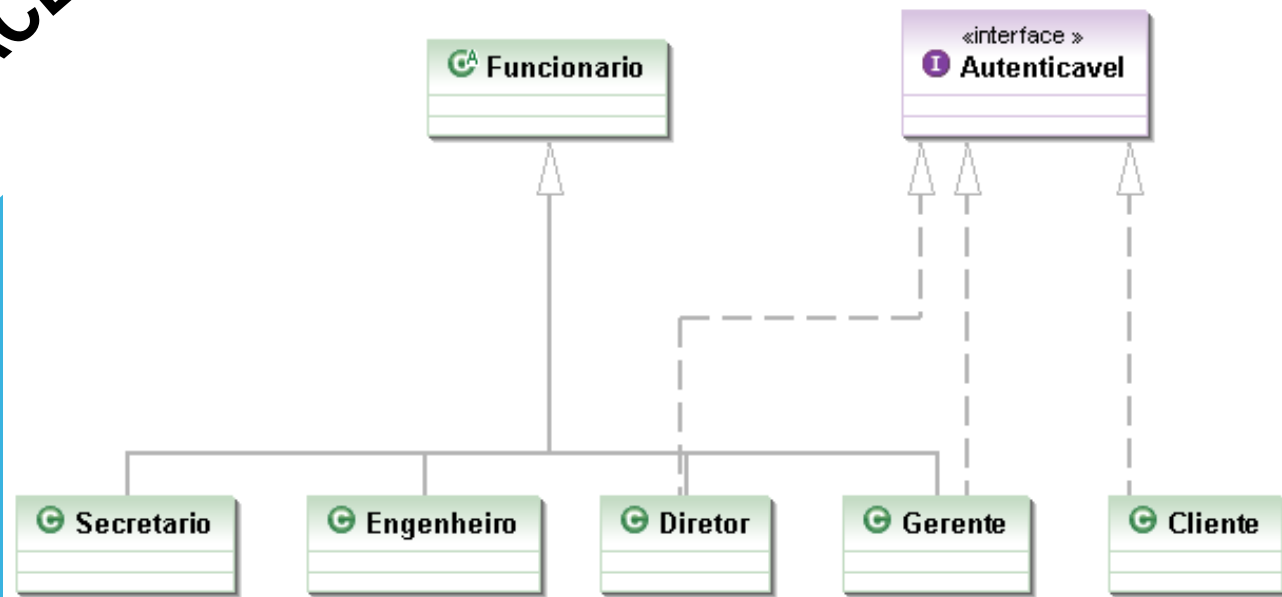
```
public abstract class ContaAbstrata {  
    private String numero;  
    private double saldo;  
  
    public ContaAbstrata(String numero){  
        this.numero = numero;  
    }  
  
    public String getNumero() {  
        return this.numero;  
    }  
  
    public double getSaldo() {  
        return this.saldo;  
    }  
  
    public void creditar(double valor) {  
        this.saldo = this.saldo + valor;  
    }  
  
    public abstract void debitar(double valor);  
}
```

June 6, 2022

MODIFICANDO A CLASSE CONTAIMPOSTO

```
public class ContaImporto extends ContaAbstrata {  
    public static final double TAXA = 0.001;  
  
    public ContaImporto (String numero) {  
        super (numero);  
    }  
  
    public void debitar(double valor) {  
        double importo = valor * TAXA;  
        double saldo = this.getSaldo();  
        if(valor + imposto <= saldo){  
            setSaldo(saldo - (valor + imposto));  
        } else {  
            System.out.println("Saldo insuficiente");  
        }  
    }  
}
```

INTERFACE



INTERFACE

- É um contrato entre a classe e o mundo externo
- Quando uma classe implementa uma interface, ela está comprometida a fornecer o comportamento publicado pela interface

INTERFACE

○ Exemplo:

Interface

```
interface IConta {  
    double saldo();  
    int numConta();  
}
```

Implementação

```
class ContaCorrente implements IConta {  
    ...  
    double saldo() {  
        // código específico  
    }  
  
    int numConta() {  
        // código específico  
    }  
    ...  
}
```

INTERFACE

○ Exemplo:

```
public interface Poligono {  
  
    public int perimetro();  
    public int area();  
  
}
```

```
public class Quadrado implements Poligono {  
    private int lado;  
  
    public int perimetro() {  
        return this.lado * 4;  
    }  
    public int area() {  
        return this.lado * this.lado;  
    }  
}
```

CLASSES ABSTRATAS X INTERFACES

Classes (abstratas)	Interfaces
Agrupar objetos com implementações compartilhadas	Agrupar objetos com implementações diferentes
Define novas classes através de herança de código	Define novas interfaces através de herança de assinaturas
Só uma classe pode ser supertipo de outra classe	Várias interfaces podem ser supertipo do mesmo tipo

AUDITOR DE BANCO DE INVESTIMENTOS

- Cliente

```
class AuditorBI {  
    final static double MINIMO = 500.00;  
    private String nome;  
    /* ... */  
  
    public boolean investigaBanco(BancoInvest b) {  
        double sm;  
        sm = b.saldoTotal() / b.numContas();  
        return (sm > MINIMO);  
    }  
}
```


AUDITOR DE BANCO DE SEGURO

○ Cliente

```
class AuditorBS {  
    final static double MINIMO = 500.00;  
    private String nome;  
    /* ... */  
  
    public boolean investigaBanco(BancoSeguros b) {  
        double sm;  
        sm = b.saldoTotal() / b.numContas();  
        return (sm > MINIMO);  
    }  
}
```

PROBLEMA

- Duplicação desnecessária de código
- O mesmo auditor deveria ser capaz de investigar qualquer tipo de banco que possua operações para calcular

DEFINIÇÃO DE INTERFACES

```
interface QualquerBanco {  
    double saldoTotal();  
    int numContas();  
}
```

UTILIZAÇÃO DE INTERFACES

○ BancoInvest

```
class BancoInvest implements QualquerBanco {  
    ...  
    double saldoTotal() {  
        /* código específico para BancoInvest */  
    }  
    int numContas() {  
        /* código específico para BancoInvest */  
    }  
    ...  
}
```

UTILIZAÇÃO DE INTERFACES

○ BancoSeguros

```
class BancoSeguros implements QualquerBanco {  
    ...  
    double saldoTotal() {  
        /* código específico para BancoSeguros */  
    }  
    int numContas() {  
        /* código específico para BancoSeguros */  
    }  
    ...  
}
```

AUDITOR GENÉRICO

- Novo cliente

```
class Auditor {  
    final static double MINIMO = 500.00;  
    private String nome;  
    /* ... */  
  
    boolean investigaBanco(QualquerBanco b) {  
        double sm;  
        sm = b.saldoTotal()/b.numContas();  
        return (sm > MINIMO);  
    }  
}
```

USANDO O AUDITOR GENÉRICO

```
QualquerBanco bi = new BancoInvest();  
QualquerBanco bs = new BancoSeguros();  
Auditor a = new Auditor();  
  
/* ... */  
boolean res1 = a.investigaBanco(bi);  
boolean res2 = a.investigaBanco(bs);  
/* ... */
```

EXCEÇÕES

```
public class TratandoExcecoesMultiCatch {  
    public void tratarExcecoes(){  
        try {  
            // Código que pode dar "erro,pau,lançar exceções,etc"  
        }catch (NumberFormatException | ArithmeticException nae) {  
            // Tratamento de 2 possíveis exceções no mesmo catch  
        }catch (Exception e) {  
            /* Tratamento para qualquer exceção  
             que não se enquadrar no catch anterior */  
        }//fecha catch  
    }//fecha método  
}
```


EXCEÇÕES

- São eventos que ocorrem durante a execução de um programa e quebram o fluxo normal de execução das instruções.
- Indicam a ocorrência de erros ou condições excepcionais no programa.

TIPOS DE EXCEÇÕES

- Erros aritméticos;
- Estouro de limite de array;
- Entrada de dados inválidos;
- Erros na manipulação de arquivos;
- Erros na comunicação com bancos de dados;
- Falhas de comunicação entre programas distribuídos;
- Entre outros.

EXEMPLO

○ Operação de Divisão

```
public class Calculadora {  
  
    public float dividir(int dividendo, int divisor){  
        return dividendo / divisor;  
    }  
}
```

Se o divisor for 0 (zero)?

EXEMPLO

- Ok, vamos resolver nosso problema:

```
public class Calculadora {  
  
    public float dividir(int dividendo, int divisor){  
  
        if(divisor != 0){  
            return dividendo / divisor;  
        }else{  
            return //?????????????????  
        }  
  
    }  
  
}
```

O quê eu retorno aqui?

PALAVRAS RESERVADAS

- Em Java:
 - **try, catch e finally**
 - Define um bloco de tratamento de exceção.
 - **throws**
 - Declara que um método pode lançar uma exceção ou mais exceções.
 - **throw**
 - Lança uma exceção.

TRATAMENTO DE EXCEÇÕES

```
public class TratandoExcecoesMultiCatch {  
    public void tratarExcecoes(){  
        try {  
            // Código que pode dar "erro,pau,lançar exceções,etc"  
        }catch (NumberFormatException | ArithmeticException nae) {  
            // Tratamento de 2 possíveis exceções no mesmo catch  
        }catch (Exception e) {  
            /* Tratamento para qualquer exceção  
             que não se enquadrar no catch anterior */  
        }//fecha catch  
    }//fecha método  
}
```

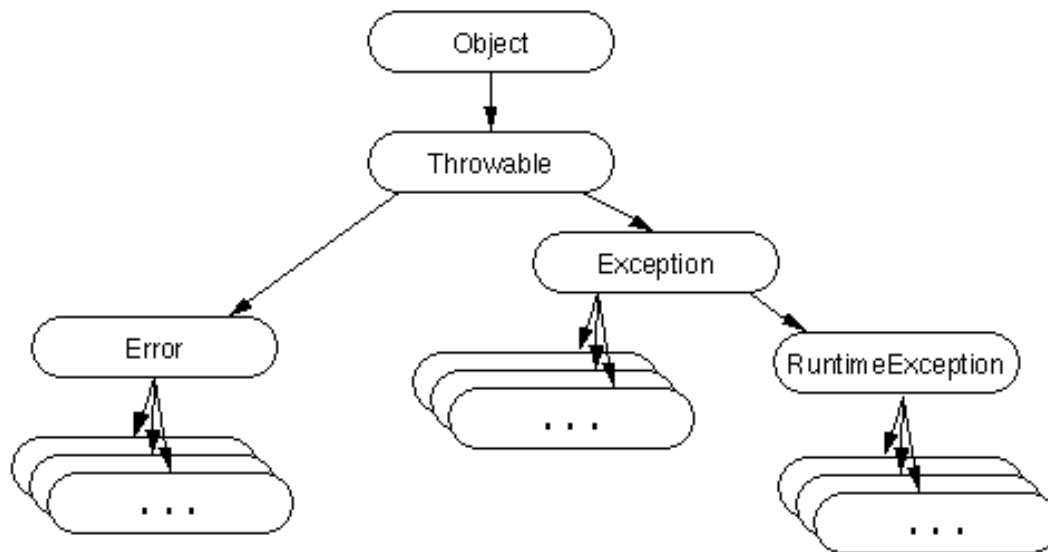
June 6, 2022

TRATAMENTO DE EXCEÇÕES

- Será que podemos tratar os erros no nosso programa antes que eles aconteçam?
 - Sim, muitas vezes podemos identificar possíveis erros e tratá-los antes mesmo da sua ocorrência.
- Exceções são objetos que encapsulam informações relevantes sobre o erro ocorrido;
- Exceções em Java são indicadas nos lugares em que elas PODEM ocorrer.

TRATAMENTO DE EXCEÇÕES

- Exceções em Java são representadas por meio de classes;
- A raiz desta hierarquia é a classe **Throwable**;



OBSERVAÇÃO: O compilador não exige que se declare ou trate exceções de qualquer subclasse de **Error** ou de **RuntimeException**.

TRATAMENTO DE EXCEÇÕES

- Classes são criadas para representar exceções específicas da aplicação em desenvolvimento;
- Um método deve informar explicitamente se pode causar uma ou mais exceções;
- Internamente a cada método, o desenvolvedor deve indicar em que situação uma condição de exceção acontece;
- Quando uma exceção acontece, algum objeto do programa precisa indicar o que deve ser feito para contornar o problema;

CRIANDO UMA EXCEÇÃO

- Toda exceção deve herdar a classe `java.lang.Exception`;

```
package ufrpe.contas;

import aula.ContaBancaria;

public class SaldoInsuficienteException extends Exception {

    private ContaBancaria conta;

    public SaldoInsuficienteException(ContaBancaria conta) {
        this.conta = conta;
    }

    public ContaBancaria getConta() {
        return this.conta;
    }

}
```

DECLARANDO EXCEÇÕES

○ Usando *throws*

- Declara que um método pode lançar uma ou mais exceções.
- Um método Java pode lançar uma exceção se encontrar uma situação com a qual ele não possa lidar;
- Um método deve informar ao compilador os parâmetros que ele recebe, o valor que ele retorna e também o que pode acontecer de errado usando *throws*.

```
public void metodo( ) throws Excecao1, Excecao2 {  
    ...  
}
```

LEVANTANDO EXCEÇÕES



- Para levantar uma exceção devemos usar a palavra *throw*

- Exemplo 1:

```
// Instanciando e lançando o objeto Exception  
throw new Exception("Mensagem de ERRO!");
```

- Exemplo 2:

```
// Instanciação do objeto Exception  
Exception e = new Exception("Mensagem de ERRO!");  
// Lançando a exceção  
throw e;
```



DECLARANDO E LEVANTANDO A EXCEÇÃO



Declarando que esse método
pode levantar uma exceção

```
public void debitar(float valor) throws SaldoInsuficienteException {  
    if (saldo < valor) {  
        throw new SaldoInsuficienteException(this);  
    }  
    saldo = saldo - valor;  
}
```

Levantando
uma exceção

TRATAMENTO DE EXCEÇÕES

- Usando *try*, *catch* e *finally*
 - Define um bloco de tratamento de exceção.

```
try {  
    //Código que pode gerar exceção  
  
} catch (Exception1 e) {  
    //Tratamento das exceções  
} catch (Exception2 e) {  
    //Tratamento das exceções  
}  
(...)  
} catch (ExceptionN e) {  
    //Tratamento das exceções  
} finally {  
    //Executa esse trecho mesmo que uma  
    //exceção ocorra.  
}
```

TRATAMENTO DE EXCEÇÕES

○ Exemplo:

```
ContaBancaria conta = new ContaBancaria("123-4");

conta.creditar(100.0);
try {
    conta.debitar(20.0f);
} catch (SaldoInsuficienteException e) {
    String mensagem = "O saldo da conta " + e.getConta().getNumero();
    mensagem += "está insuficiente";
    System.out.println(mensagem);
} finally{
    System.out.println("Fim da operação!");
}
```



PROGRAMAÇÃO ORIENTADA A OBJETO - JAVA

PROFESSOR: DANILO FARIAS
SEMESTRE: 2020.1

June 6, 2022