

PROGRAMAÇÃO ORIENTAÇÃO A OBJETOS

Jadir Custódio Mendonça Junior

Programação Orientada a Objetos



Programação Orientada a Objetos

O que é Orientação a Objetos

- A orientação a Objetos visa **abstrair** características de entidades concretas e abstratas do mundo real, criando padrões ou classificações que posteriormente serão transformados em objetos virtuais.

Programação Orientada a Objetos

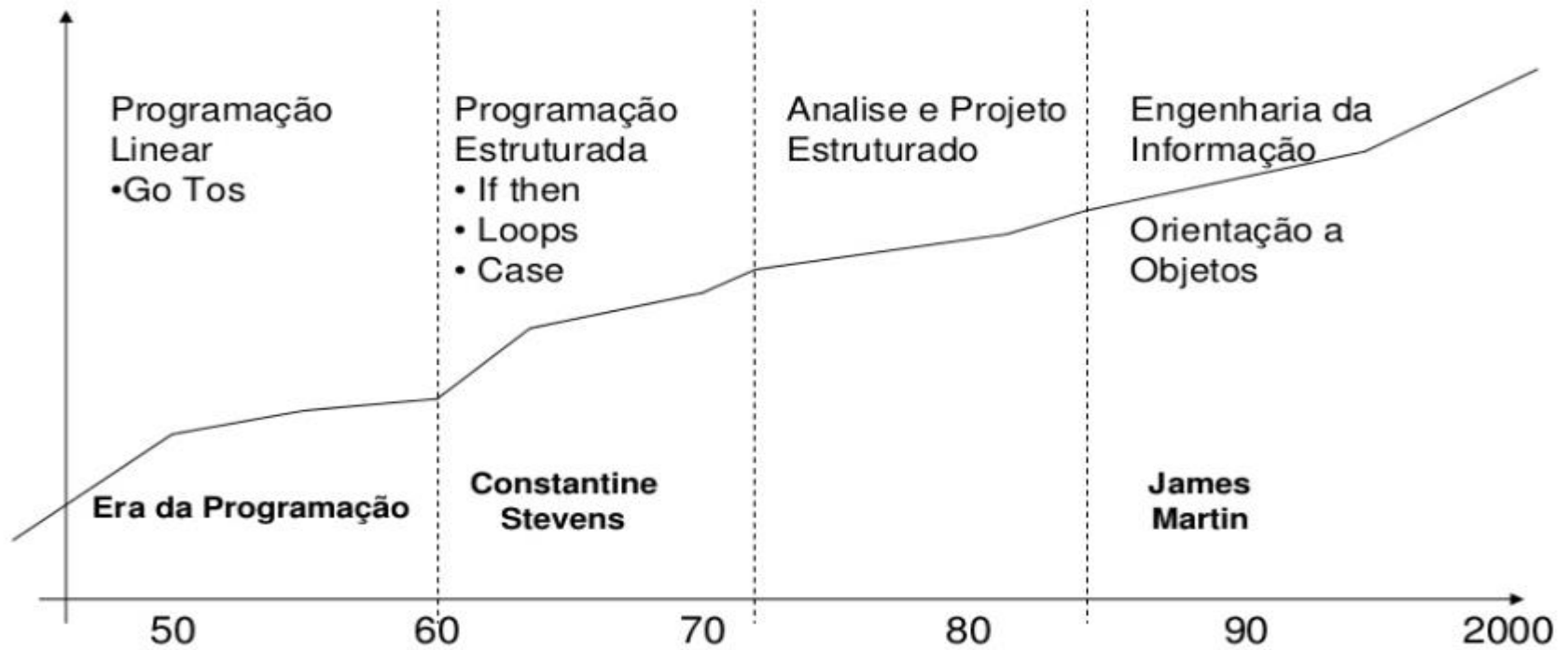


O QUE É POO E
PORQUE VOCÊ
PRECISA
APRENDER!

Be

Princípios do Paradigma de O.O.

Evolução



Linguagens de Programação O.O.

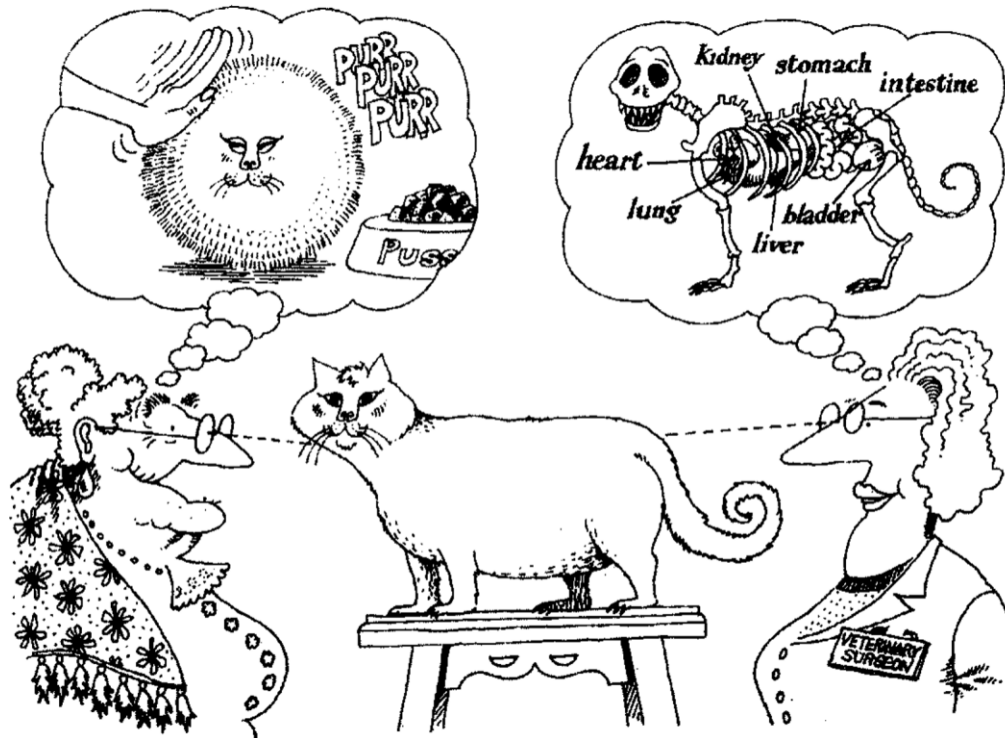


Princípios do Paradigma de O.O.



Abstração

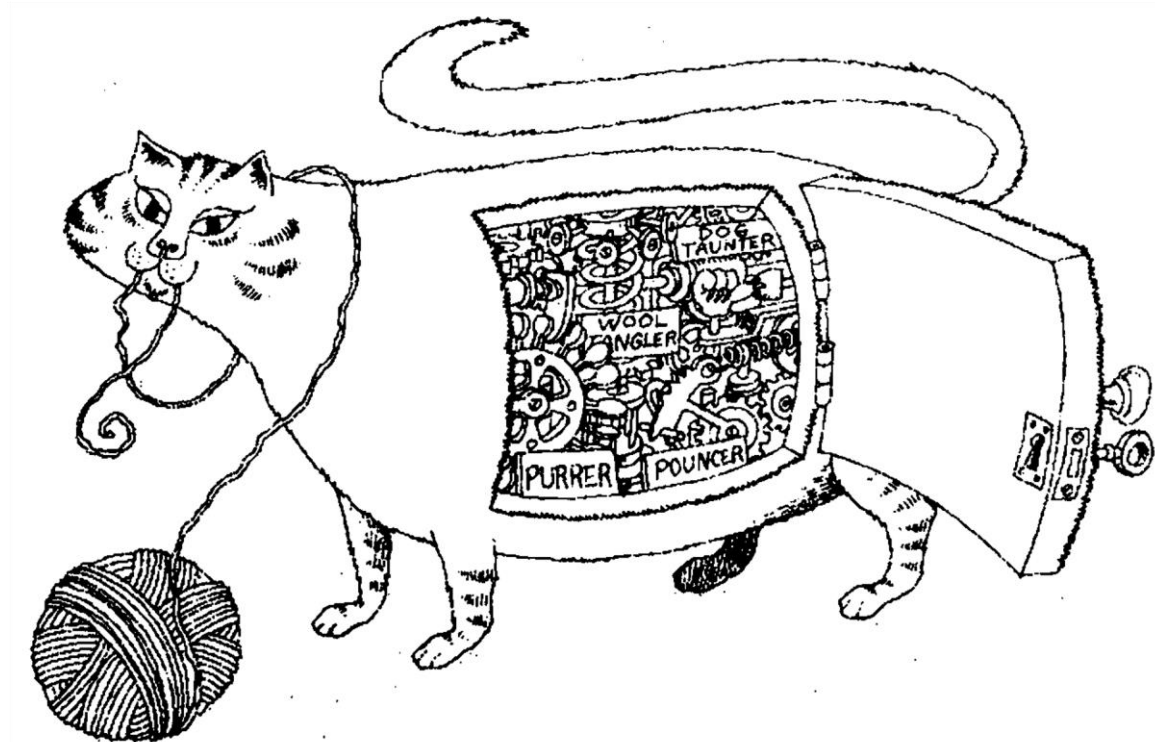
- A representação computacional do objeto real deve se concentrar nas características que são relevantes para o problema



Fonte: livro “Object-Oriented Analysis and Design with Applications”

Encapsulamento

- O objeto deve esconder seus dados e os detalhes de sua implementação



Fonte: livro “Object-Oriented Analysis and Design with Applications”

Modularidade

- O sistema deve ser composto de objetos **altamente coesos** e **fracamente acoplados**



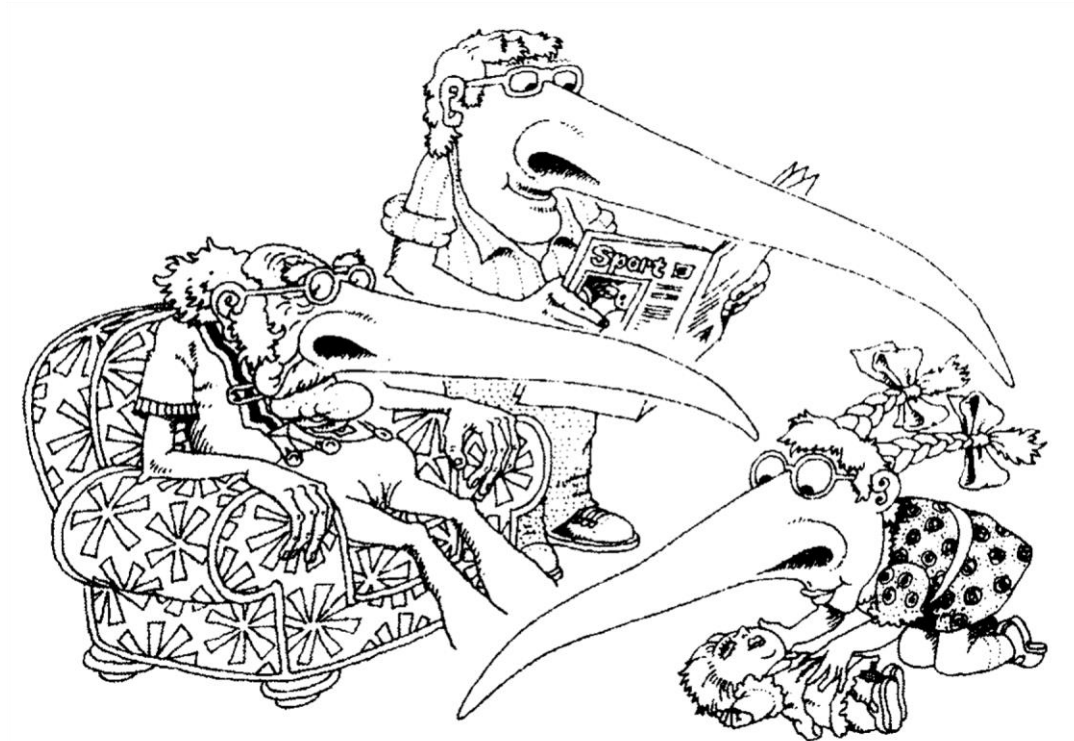
Fonte: livro “Object-Oriented Analysis and Design with Applications”

Reutilização – o pulo do Gato



Hierarquia

- Objetos herdam atributos e métodos dos seus ancestrais na hierarquia

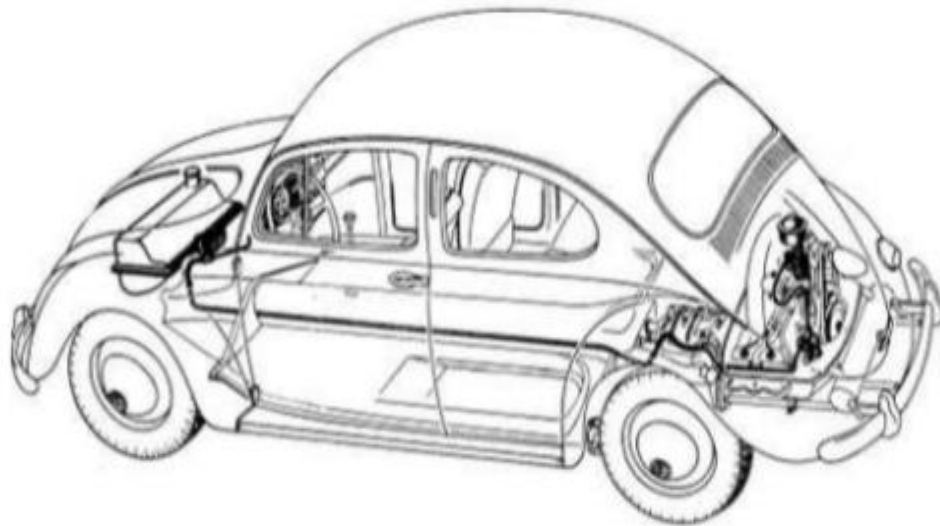


Fonte: livro “Object-Oriented Analysis and Design with Applications”

Classe

- **Classe** é um conjunto de objetos com características e comportamentos afins.

SISTEMA DE ALIMENTAÇÃO



Classe

- Representa um conjunto de objetos com a mesma estrutura;
- Modelo do qual objetos individuais são criados;
- Classes devem representar uma abstração.


Estrutura de uma classe

- Toda a classe tem um nome que a identifica
 - ▣ Em Java, o nome da classe deve sempre começar com a primeira letra maiúscula:
 - Exemplo.:
 - Pessoa, Automovel, Conta,...
 - ▣ No caso de nomes compostos utilize letra maiúscula nos dois:
 - Exemplo.:
 - ContaCorrente, ApoliceSeguro,...

Objeto

- **Objeto** é um elemento concreto de um tipo de classe.

Fusca (1) Fusca (2) Fusca (3)



Estrutura de uma classe

- Uma classe define **CARACTERÍSTICAS** e **COMPORTAMENTO** de um objeto;
- Em Java, define-se **características** e **comportamentos** de uma classe através de **atributos** e **métodos**;
- **Atributos (características):**
 - ▣ São possíveis dados armazenados por um objeto de uma classe representando o seu estado.
- **Métodos (comportamentos):**
 - ▣ São procedimentos que formam os comportamentos e serviços oferecidos por objetos de uma classe.

Exercício

- Faça a definição de classe de 3 objetos **incomuns**, abstraindo ao menos 3 propriedades e 2 métodos de cada um.
- **Desafio** : Crie classes no java para a definição destes objetos.
- **Dica** : Documente primeiro e codifique depois, se não vale a pena ser documentado é por que não vale a pena ser feito.

Estrutura de uma classe

- O código abaixo mostra como criar uma classe em Java:
 - ▣ A classe ContaCorrente:

```
public class ContaCorrente {  
    /*  
     * É dentro deste escopo que definimos  
     * toda a estrutura da classe como atributos,  
     * métodos .  
     */  
}
```

Estrutura de uma classe

- O código abaixo mostra classe Agencia com seus atributos:

```
public class Agencia {  
    //Definição de Atributos  
    int nrAgencia;  
    String nomeAgencia;  
}
```

Estrutura de uma classe

- O código abaixo mostra classe Cliente com seus atributos:

```
public class Cliente {  
    //Definição de Atributos  
    String cpf;  
    String nomeCliente;  
}
```

Estrutura de uma classe

- O código abaixo mostra classe Conta Corrente com seus atributos:

```
public class ContaCorrente {  
    //Definição de Atributos  
    int conta;  
    int agencia;  
    double saldo;  
    String nome;  
}
```

Estrutura de um método

Em Java, todo método possui uma declaração e um corpo, cuja estrutura simplificada é formada por:

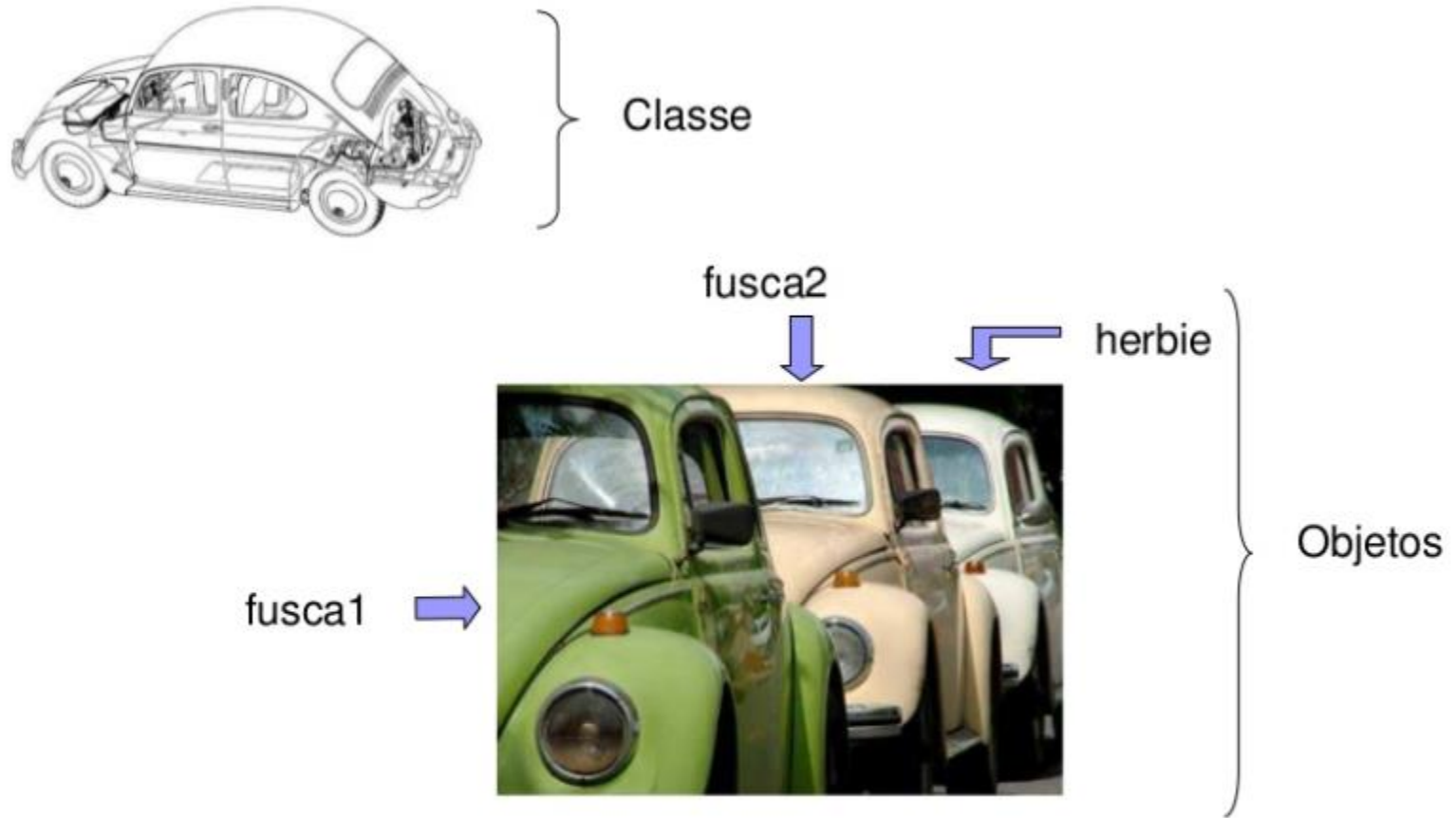
```
qualificador tipo-do-retorno nome-do-método ([lista-de-parametros])  
{  
    [escopo-de-instruções]  
}
```

Sendo que:

Estrutura de uma classe

```
public class ContaCorrente {  
    //Definição de Atributos  
    int conta;  
    int agencia;  
    double saldo;  
    String nome;  
    /* Definição dos métodos */  
    public void depositar(double valor) {  
        saldo = saldo + valor;  
    }  
    public void sacar(double valor) {  
        saldo = saldo - valor;  
    }  
    public double imprimirSaldo() {  
        return saldo;  
    }  
    public void imprimirDados() {  
        System.out.println(nome, agencia, saldo, nome);  
    }  
}
```


Instância



- A partir de uma mesma classe é possível criar diversos objetos.

Estrutura de um método

```
qualificador tipo-do-retorno nome-do-método ([lista-de-parametros])  
{  
    [escopo-de-instruções]  
}
```

Qualificador: conhecido também como **modificador de acesso**, define a visibilidade do método. Trata-se de uma forma de especificar se o **método é visível** apenas dentro da própria classe em que se está declarado, ou pode ser visualizado (e utilizado) por classes externas. O qualificador do método pode ser do tipo:

- ▣ **public**: o método é visível por qualquer classe. É o qualificador mais aberto no sentido de que qualquer classe pode usar esse método;
- ▣ **private**: o método é visível apenas pela própria classe. É o qualificador mais restritivo;
- ▣ **protected**: o método é visível pela própria classe, por suas subclasses e pelas classes do mesmo pacote.

Veremos mais detalhes sobre qualificadores em **Encapsulamento**.

Estrutura de um método

```
qualificador tipo-do-retorno nome-do-método ([lista-de-parametros])  
{  
    [escopo-de-instruções]  
}
```

Tipo-de-retorno: refere-se ao tipo de dado retornado pelo método. Métodos que não retornam valores devem possuir nesse parâmetro a palavra **void**. Sempre que **void** for usada em uma declaração de método, nenhum valor será retornado após a sua execução, isto é, o método atua como uma *procedure* (procedimento) como em outras linguagens de programação.

- Um método pode ter como retorno:
 - ▣ Tipos primitivos: byte, int, short, long, float, double e char;
 - ▣ Vetores (Arrays): int[], String[], List[],...
 - ▣ Tipos de referência (Tipos Objetos): String, Scanner, Aluno.

Estrutura de um método

```
qualificador tipo-do-retorno nome-do-método ([lista-de-parametros])  
{  
    [escopo-de-instruções]  
}
```

Nome-do-método: pode ser qualquer palavra ou frase, desde que iniciada por uma letra. Se o nome for uma frase, não podem existir espaços em branco entre as palavras. Como padrão Java, o nome de um método **sempre inicia com uma palavra com letras minúsculas**. Caso o nome do método seja composto, a partir do segundo nome deve iniciar com letra maiúscula:

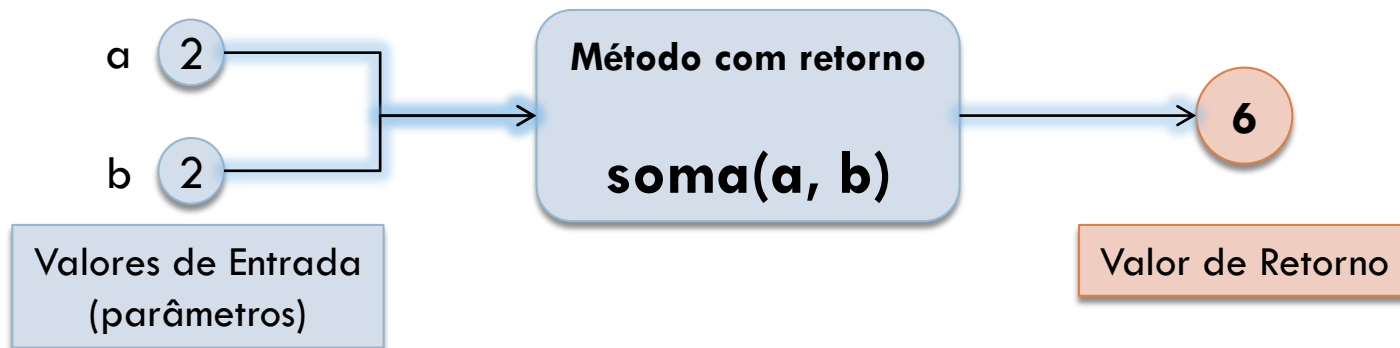
□ Exemplo de nome de métodos:

- imprimir, sacar, depositar,...
- imprimir**D**ados, gravar**A**rquivo, set**N**ome, get**I**dade,...
- emitir**N**ota**F**iscal, set**N**ome**C**liente, exec**A**tualizar**S**aldo,...

Estrutura de um método

```
qualificador tipo-do-retorno nome-do-método ([lista-de-parâmetros])  
{  
    [escopo-de-instruções]  
}
```

Lista-de-parâmetros: trata se de uma lista de variáveis opcionais, que podem ser recebidas pelo método para tratamento interno. Quando um método é invocado (chamado), **ele pode receber valores de quem o chamou**. Esses valores podem ser manipulados internamente e devolvidos ao **emissor da solicitação**. Abaixo apresenta uma analogia exemplificando uma chamada a um método com **entrada de parâmetros** e **valor de retorno**:



Estrutura de um método

```
qualificador tipo-do-retorno nome-do-método ([lista-de-parametros])  
{  
    [escopo-de-instruções]  
}
```

Escopo-de-instruções: trata-se dos códigos implementados em Java que realizam os processos internos e retornam os valores desejados, isto é, constituem o programa do método, por exemplo:

```
{  
    int valorA, valorB, soma;  
  
    valorA = a;  
    valorB = b;  
    soma = valorA + valorB;  
    System.out.println("A Soma é: " + soma);  
    ...  
}
```



Escopo de Instruções

Método sem retorno de valor

- Métodos sem retorno são definidos como **void**.

```
class ContaCorrente {  
    /* Definição de atributos...*/  
  
    /* Definição dos métodos */  
  
    public void depositar(double valor) {  
        saldo saldo + valor;  
    }  
  
    public void sacar(double valor) {  
        saldo = saldo - valor;  
    }  
}
```

Método com retorno de valor

- São métodos que retornam valores de acordo o tipo de dado retorno.

```
public class ContaCorrente {  
    /* Definição de Atributos ... */  
  
    /* Definição dos métodos */  
    public double getSaldo() {  
        return saldo;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
}
```


Objetos em Java

- Objeto é a instância de uma classe, ou seja, é a concretização de uma classe.
- Um objeto é algo que existe fisicamente e que foi "moldado" na classe.
- A criação de um objeto em Java é feita conforme comando abaixo:
 - ▣ `Classe variavel = new Classe();`
 - ▣ Onde:
 - **Classe:** é o tipo de variável Objeto;
 - **variavel:** é o nome da **variável objeto** propriamente dita;
 - **new:** é o operador de criação da variável objeto (alocação de memória, chamada do construtor da classe);
 - **Classe():** é o **método construtor da classe**, responsável por construir um objeto com determinados valores.

Criação/Instância de Objetos

O programa abaixo apresenta a criação e/ou instanciação de objetos do tipo `ContaCorrente`:

```
public class TesteConta {  
    public static void main(String args[]) {  
        //Criando o objeto conta1  
        ContaCorrente conta1 = new ContaCorrente();  
  
        //Criando o objeto conta2  
        ContaCorrente conta2 = new ContaCorrente();  
  
        //Criando o objeto conta3  
        ContaCorrente conta3 = new ContaCorrente();  
    }  
}
```

Criação/Instância de Objetos

O programa abaixo apresenta a criação e/ou instanciação de um objeto do tipo **ContaCorrente** bem como atribuição de valores e chamada à métodos:

```
public class TesteConta {  
    public static void main(String args[]) {  
        //Criando o objeto conta  
        ContaCorrente conta = new ContaCorrente();  
  
        //atribuição de valores:  
        conta.nome = "Maria João";  
        conta.agencia = 2045;  
        conta.conta = 3132547;  
  
        //chamada a métodos:  
        conta.depositar(300.50);  
        conta.sacar(150.00);  
        // será impresso o nome do cliente e o seu saldo (150.50)  
        conta.imprimeDados();  
    }  
}
```

Exercícios Práticos

- 1-a) Criar uma classe chamada **Computador** com as seguintes definições:
 - ▣ atributos: **marca**, **cor**, **modelo**, **serie** e **valor**;
 - ▣ Definir o método **imprimirDados()** de forma que imprima os dados advindos dos atributos.
- 1-b) Criar uma classe chamada Principal com a seguinte estrutura:
 - ▣ Criar o método **main()** conforme o padrão do Java;
 - ▣ Criar um objeto da classe **Computador** e atribuir valores aos seus atributos. Atribuir “HP” ao atributo **marca**;
 - ▣ Executar o método **imprimirDados()** para exibir os dados.

Exercícios Práticos

- 2-a) Criar uma classe chamada **Eleitoral** com as seguintes definições:
 - ▣ atributos: **nome**, e **idade**;
 - ▣ Definir os seguintes métodos:
 - **imprimir()** : O método deverá imprimir os dados dos atributos nome e idade, e deverá executar o método verificar() como último comando.
 - **verificar()** : O método **não retorna valor** nem **recebe parâmetro**. Deve exibir na tela mensagens de acordo com as seguintes condições: caso a idade **seja inferior a 16 anos**, exibir na tela “○ **Eleitor não pode votar**”. Para idade **superior ou igual a 16 e inferior ou igual 65**, exibir “○ **Eleitor deve Votar**”. Para **maiores de 65 anos**, exibir “**Voto facultativo**”

Exercícios Práticos

- 2-b) Cria uma classe chamada **PrincipalEleitoral** com a seguinte estrutura:
 - ▣ Criar o método `main()` conforme o padrão da linguagem Java;
 - ▣ Criar um objeto da classe **Eleitoral** e atribuir valores e atribuir valores aos parâmetros;
 - ▣ Executar o método `imprimir()` para exibir os valores.
 - ▣ Executar o método `verificar()` para saber se o eleitor pode VOTAR.

Exercícios Práticos

- 3-a) Criar uma classe que simule o funcionamento de uma lâmpada. A classe **Lampada** deve conter um atributo booleano chamado **status** e os métodos **ligar()** e **desligar()** (ambos sem retorno). O método **ligar()** atribui o valor **true** ao atributo **status**, já o método **desligar()** atribui o valor **false** à **status**. Crie também o método **observar()** que retorna uma palavra referente ao estado da lâmpada (“Ligada” ou “desligada”).
- 3-b) Crie uma classe chamada **UsaLampada** que utilize a classe **Lampada** do exercício anterior. Ela deve conter o método **main()** e:
 - ▣ Instanciar dois objetos da classe **Lampada** (**lampada1** e **lampada2**);
 - ▣ Ligar o objeto **lampada1** e desligar o objeto **lampada2**;
 - ▣ Executar o método **observar()** para exibir o status das lâmpadas.

Construtores

40

- Em Java, o método construtor é definido como um método cujo o nome deve ser o **mesmo nome da classe** e ainda sem indicação do tipo de retorno;
- Um construtor é unicamente **invocado (chamado)** no momento da **criação (instanciação)** do objeto por meio do operador **new**;
- Uma classe pode ter um ou vários métodos construtores definidos em sua estrutura:
 - ▣ **O que diferencia** um método construtor de outro na mesma classe **são os seus parâmetros**.
- É importante observar que, quando **não criamos nenhum construtor de forma explícita**, o compilador Java **cria um automaticamente**:
 - ▣ Construtor vazio (sem parâmetros e sem argumentos)

Construtores

41

Exemplo de criação e uso de classe com construtor:

```
public class Aluno {  
    //Atributos  
    String nome; int rgm;  
  
    //Método Construtores  
    public Aluno() {  
  
    }  
  
    public Aluno(String nome, int rgm) {  
        this.nome = nome;  
        this.rgm = rgm;  
    }  
  
    //Outros métodos...  
}
```

```
public class Principal {  
    public static void main(String[] args) {  
        /* Criando um objeto com  
         * construtor sem parametros */  
        Aluno aluno1 = new Aluno();  
  
        /* Criando um objeto com  
         * construtor com parametros */  
        Aluno aluno2 = new Aluno("Almir", 12510750);  
    }  
}
```

Encapsulamento - Conceitos

42

- Encapsulamento é um mecanismo que possibilita **restringir o acesso** à **atributos** e **métodos** de um **objeto**.
- Encapsulamento *(também chamado ocultamento de informações)* separa os **aspectos externos** dos **detalhes internos** da implementação de um **objeto**.

Encapsulamento - Conceitos

43

- O encapsulamento promove e facilita a manutenção do programa:
 - ▣ Podemos mudar a implementação de um objeto sem afetar as aplicações que o utilizam;
 - ▣ Podemos querer mudar a implementação de um objeto para melhorar o seu desempenho;
 - ▣ Reparar um erro, consolidar código ou dar suporte a portabilidade.

Encapsulamento - Conceitos

44

- Algumas características do Encapsulamento:
 - ▣ Um objeto **não pode manipular os atributos** de outro objeto **diretamente**;
 - ▣ Um objeto apenas pode acessar atributos de outro objeto mediante ao **envio de mensagens**;
 - ▣ A **interação entre objetos** se dá apenas por meio de **mensagens**;
 - ▣ A única coisa que um objeto conhece sobre outro objeto é a interface;
 - ▣ A Interface de um objeto encapsula os seus dados e código.
- Entenda por interface, os métodos de uma classe que representam os comportamentos de um objeto.

Encapsulamento - Mensagens

45

- O que são Mensagens?
 - ▣ Uma mensagem é uma forma de comunicação e interação entre objetos em uma aplicação;
 - ▣ Em outras palavras, um **objeto A** pode necessitar de um procedimento (método) já definido em um **objeto B**;
 - ▣ Para realizar esse processo, o **objeto A** solicita ao **objeto B** que execute um método;
- Uma mensagem nada mais é do que o fato de um objeto chamar o método de um outro objeto:
 - ▣ Em orientação a objetos, quando um **objeto A** solicita a um **objeto B** que execute um de seus métodos, diz-se que o **objeto A** enviou uma mensagem para o **objeto B**

Encapsulamento - Mensagens

46

Abaixo temos um exemplo de interação de objetos

```
public class ContaCorrente {  
    // Definição do atributo saldo  
    double saldo = 300.0;  
  
    /* Definição dos método sacar */  
    void efetuarSaque(double valor) {  
        this.saldo -= valor;  
    }  
}
```

```
public class Principal {  
    public static void main(String[] args){  
        ContaCorrente conta1 new ContaCorrente();  
  
        // enviando uma mensagem ao objeto1...  
        conta1.efetuarSaque(150.0);  
    }  
}
```

Encapsulamento - Mensagens

47

Um objeto da classe **Principal** foi criado pelo JVM!

```
public class ContaCorrente {  
    // Definição do atributo saldo  
    double saldo = 300.0;  
  
    /* Definição dos método sacar */  
    void efetuarSaque(double valor) {  
        this.saldo -= valor;  
    }  
}
```

```
public class Principal {  
    public static void main(String[] args){  
        ContaCorrente conta1 new ContaCorrente();  
  
        // enviando uma mensagem ao objeto1...  
        conta1.efetuarSaque(150.0);  
    }  
}
```



criado pela
JVM

Objeto
(Principal)

Encapsulamento - Mensagens

48

O objeto da classe principal criou um outro objeto de **ContaCorrente**!

```
public class ContaCorrente {  
    // Definição do atributo saldo  
    double saldo = 300.0;  
  
    /* Definição dos método sacar */  
    void efetuarSaque(double valor) {  
        this.saldo -= valor;  
    }  
}
```

```
public class Principal {  
    public static void main(String[] args){  
        ContaCorrente conta1 new ContaCorrente();  
  
        // enviando uma mensagem ao objeto1...  
        conta1.efetuarSaque(150.0);  
    }  
}
```

Linha que cria um
objeto da classe
ContaCorrente

criado pela
JVM

Objeto
(Principal)

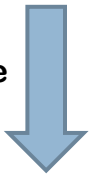
Encapsulamento - Mensagens

49

O objeto da classe principal criou um outro objeto de **ContaCorrente**!

```
public class ContaCorrente {  
    // Definição do atributo saldo  
    double saldo = 300.0;  
  
    /* Definição dos método sacar */  
    void efetuarSaque(double valor) {  
        this.saldo -= valor;  
    }  
}
```

criado pelo
objeto da classe
Principal



Conta1
(ContaCorrente)

```
public class Principal {  
    public static void main(String[] args){  
        ContaCorrente conta1 new ContaCorrente();  
  
        // enviando uma mensagem ao objeto1...  
        conta1.efetuarSaque(150.0);  
    }  
}
```

criado pela
JVM



Objeto
(Principal)

Encapsulamento - Mensagens

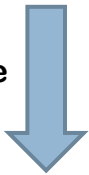
50

Objeto da classe principal envia uma mensagem ao objeto de **ContaCorrente...**

```
public class ContaCorrente {  
    // Definição do atributo saldo  
    double saldo = 300.0;  
  
    /* Definição dos método sacar */  
    void efetuarSaque(double valor) {  
        this.saldo -= valor;  
    }  
}
```

```
public class Principal {  
    public static void main(String[] args){  
        ContaCorrente conta1 new ContaCorrente();  
  
        // enviando uma mensagem ao objeto1...  
        conta1.efetuarSaque(150.0);  
    }  
}
```

criado pelo
objeto da classe
Principal



criado pela
JVM



Conta1
(ContaCorrente)

Objeto
(Principal)



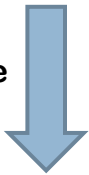
Encapsulamento - Mensagens

51

Chamando o método **efetuarSaque()**!

```
class ContaCorrente {  
    // Definição do atributo saldo  
    double saldo = 300.0;  
  
    /* Definição dos método sacar */  
    void efetuarSaque(double valor) {  
        this.saldo -= valor;  
    }  
}
```

criado pelo
objeto da classe
Principal



Conta1
(ContaCorrente)

```
public class Principal {  
    public static void main(String[] args){  
        ContaCorrente conta1 new ContaCorrente();  
  
        // enviando uma mensagem ao objeto1...  
        conta1.efetuarSaque(150.0);  
    }  
}
```

criado pela
JVM



Objeto
(Principal)

mensagem: efetuarSaque(150.0)



Encapsulamento - visibilidade

52

- Cada elemento que constitui uma classe possui uma visibilidade;
- Visibilidade é a capacidade de um objeto poder "ver" ou de ter referência a outro objeto;
- O conceito de visibilidade também é conhecido como modificadores de acesso.

Encapsulamento - visibilidade

53

- Para determinar o nível de acesso dos elementos de uma classe em Java, usamos os chamados qualificadores de acesso, também conhecidos como modificadores de acesso:
 - **public:** um nível sem restrições, equivalente a não encapsular, ou seja, se um atributo for definido como **public**, não será possível realizar o encapsulamento;
 - **private:** o nível de maior restrição em que apenas a própria classe pode ter acesso a atributos e/ou métodos;
 - **protected:** um nível intermediário de encapsulamento em que as variáveis e métodos podem ser acessados pela própria classe ou por suas subclasses;
 - **package:** nível em que a classe pode ser acessada apenas por outras classes pertencentes ao mesmo pacote.

Encapsulamento - visibilidade

54

- A visibilidade é aplicada **as classes**, de forma que:
 - ▣ Uma classe **seja** ou **não** visível a outra classe armazenadas no mesmo ou em outros pacotes;
- A visibilidade é aplicada **aos membros** de uma classe, de forma que:
 - ▣ Os Atributos deva somente ser acessados pelos métodos da classe a que pertence, ou seja, os atributos devem ser declarados privados (**private**);
 - ▣ Os métodos devem ser declarados públicos (**public**), pois são os responsáveis pela modificação de dados dos atributos e servem como interface aos usuários da classe.

Métodos Getters e Setters

55

- Os **Getters** e **Setters** são métodos definidos em uma classe que provê o acesso e modificação de atributos de atributos do objeto desta classe;
- São os **Getters** e **Setters** que garantem o encapsulamento dos atributos:
 - ▣ Evita que os atributos sejam acessados de forma direta.
- Esse encapsulamento oferece uma proteção ao atributo para garantir que os valores atribuídos estejam dentro dos valores aceitáveis para o atributo;

Métodos Getters e Setters

56

Vejam os um exemplo da classe **Data** conforme a definição abaixo:

```
public class Data {  
    // definição de atributos  
    int dia;  
    int mes;  
    int ano;  
}
```


Métodos Getters e Setters

57

Veja que a classe **Data** está sendo usada na classe **Principal** de forma direta...

```
public class Data {  
    // definição de atributos  
    int dia;  
    int mes;  
    int ano;  
}
```

```
public class Principal {  
    public static void main(String[] args) {  
        //criando e usando a classe Data  
        Data dta = new Data();  
        dta.dia = 12;  
        dta.mes = 7;  
        dta.ano = 1982;  
    }  
}
```

Métodos Getters e Setters

58

Mas, quem garante o que o valor atribuído aos atributos serão coerentes?

```
public class Data {  
    // definição de atributos  
    int dia;  
    int mes;  
    int ano;  
}
```

```
public class Principal {  
    public static void main(String[] args) {  
        //criando e usando a classe Data  
        Data dta = new Data();  
        dta.dia = 35; //dia inválido  
        dta.mes = 15; //mês inválido  
        dta.ano = -1983; //ano inválido  
    }  
}
```

Métodos Getters e Setters

59

- Para resolver este problema, utilizamos os métodos **Getters** e **Setters** para validar os dados dos seus atributos de forma que:
 - ▣ Cada método **Setter** seja responsável por garantir a validação e integridade do dado inserido;
 - ▣ Cada método **Getter** seja responsável por trazer o dado correto de um atributo;
- Os **Getters** e **Setters** devem ser declarados como públicos, pois definem uma interface ao usuário da classe.

Veja um esboço da classe **Data** com os métodos **getters** e **Setters** no próximo slide...

```
public class Cliente {  
    // atributo  
    String cpfCliente;  
    String nomeCliente;  
    // getters e setters  
    public String getCpfCliente() {  
        return cpfCliente;  
    }  
    public void setCpfCliente(String cpfCliente) {  
        this.cpfCliente = cpfCliente;  
    }  
    public String getNomeCliente() {  
        return nomeCliente;  
    }  
    public void setNomeCliente(String nomeCliente) {  
        this.nomeCliente = nomeCliente;  
    }  
}
```

Métodos Getters e Setters

61

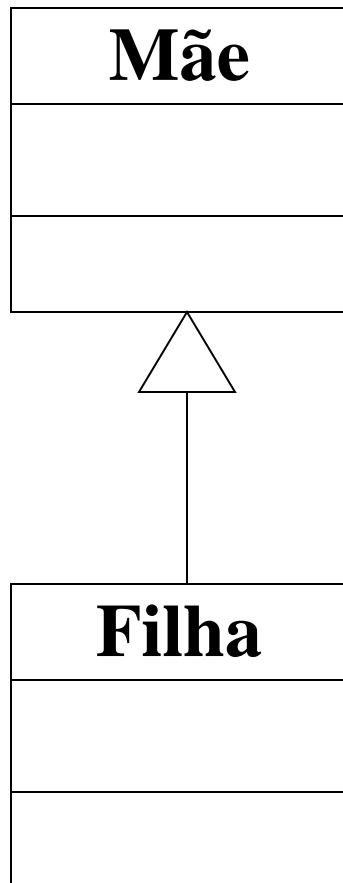
Abaixo mostra a criação e uso de um objeto da classe **Cliente** utilizando os métodos **Getters** e **Setters**

```
public class Principal {  
    public static void main(String[] args) {  
        //criando e usando a classe Cliente  
        Cliente clie = new Cliente();  
  
        //Uso dos Setters  
        clie.setCpfCliente("092.934.945-94");  
        clie.setNomeCliente("Luis Felipe");  
  
        //Uso dos Getters  
        System.out.println(clie.getCpfCliente());  
        System.out.println(clie.getNomeCliente());  
    }  
}
```

Herança

- É a capacidade de um novo objeto tomar atributos e operações de um objeto existente, permitindo criar classes complexas sem repetir código.
- A nova classe simplesmente herda seu nível base de características de um antepassado na hierarquia de classe.

Herança

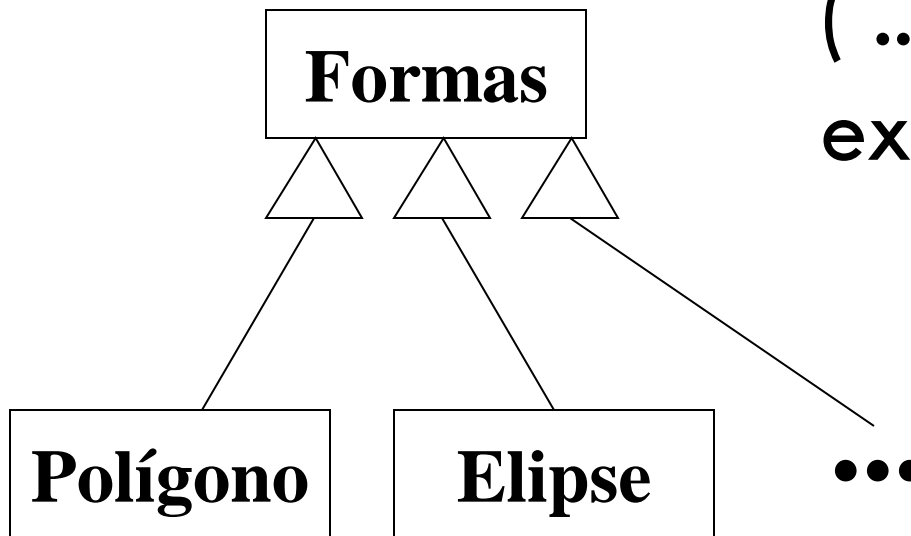


- ❑ A classe “filha” herda os atributos, operações e relacionamentos da classe “mãe”
- ❑ Não existe nome nem multiplicidade na relação

Herança

- **Notação de herança na UML**

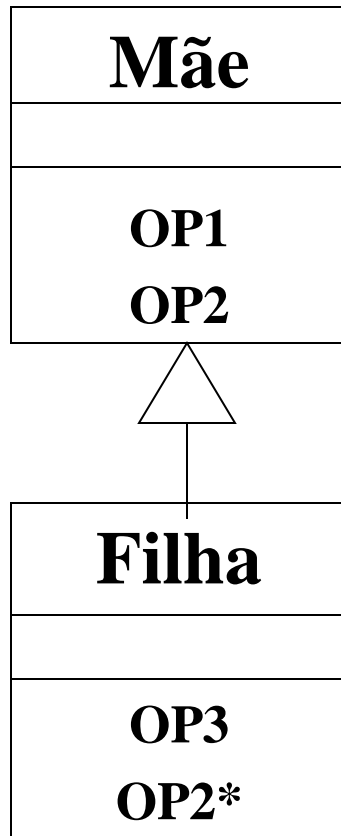
subtipo : supertipo



(...) Indica que
existem outras
subclasses

Herança

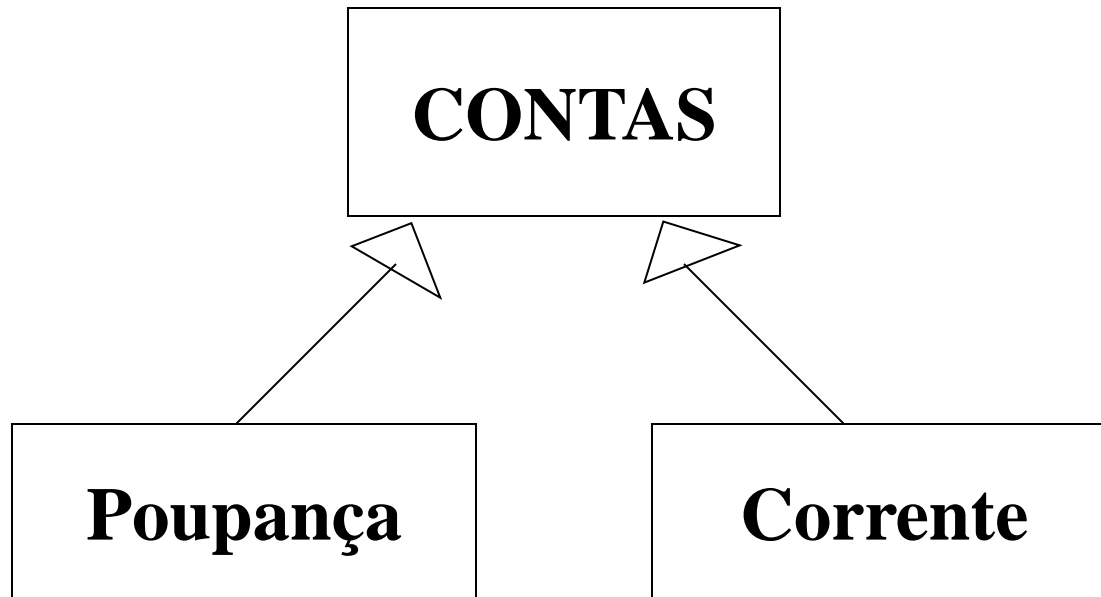
65



- A classe Mãe possui as operações OP1 e OP2
- A classe filha possui as operações OP1, OP2* e OP3
- A operação OP2 (mesmo nome) é sobrescrita (override) na classe filha

Exemplo de Herança

66



Exemplo de Herança

```
Class Veiculo{  
    String nome;  
    float velocidade;  
    public void acelera() {  
        if (velocidade<=10)  
            velocidade++;  
    }  
    public void frea() {  
        if (velocidade>0)  
            velocidade--;  
    }  
}
```

**Veiculo2 estende
Veiculo**

```
Class Veiculo2 extends Veiculo{  
    boolean ligado;  
    public void liga() {  
        ligado = true;  
    }  
    public void desliga() {  
        ligado = false;  
    }  
}
```

Exemplo de Herança

```
class Generica{  
    public static void main ( String args[]) {  
        // Cria um objeto da classe Veiculo2  
        Veiculo2 A = new Veiculo2();  
        // Cria um veiculo da classe Veiculo  
        Veiculo B = new Veiculo();  
        A.liga();  
        A.acelera();  
        B.acelera();  
        A.desliga();  
    }  
}
```

Exercício de Herança

1. Crie uma classe **Animal** que obedeça à seguinte descrição:
 - possua os atributos **nome** (String), **comprimento** (float), **número de patas** (int), **cor** (String), **ambiente** (String) e **velocidade média** (float)
 - Crie um método construtor que receba por parâmetro os valores iniciais de cada um dos atributos e atribua-os aos seus respectivos atributos.
 - Crie os métodos **get** e **set** para cada um dos atributos.
 - Crie um método **dados**, sem parâmetro e do tipo **void**, que, quando chamado, imprime na tela uma espécie de relatório informando os dados do animal.

Exercício de Herança

2. Crie uma classe **Peixe** que herde da classe **Animal** e obedeça à seguinte descrição:
 - possua um atributo **caracteristica**(String)
 - Crie um método construtor que receba por parâmetro os valores iniciais de cada um dos atributos (incluindo os atributos da classe **Animal**) e atribua-os aos seus respectivos atributos.
 - Crie ainda os métodos **get** e **set** para o atributo **caracteristica**.
 - Crie um método **dadosPeixe** sem parâmetro e do tipo void, que, quando chamado, imprime na tela uma espécie de relatório informando os dados do peixe (incluindo os dados do Animal e mais a característica).

Exercício de Herança

3. Crie uma classe **Mamifero** que herde da classe **Animal** e obedeça à seguinte descrição:
 - possua um atributo **alimento**(String)
 - Crie um método construtor que receba por parâmetro os valores iniciais de cada um dos atributos (incluindo os atributos da classe **Animal**) e atribua-os aos seus respectivos atributos.
 - Crie ainda os métodos **get** e **set** para o atributo **alimento**.
 - Crie um método **dadosMamifero** sem parâmetro e do tipo void, que, quando chamado, imprime na tela uma espécie de relatório informando os dados do mamifero (incluindo os dados do Animal e mais o alimento).

Exercício de Herança

4. Crie uma classe **TestarAnimais** possua um método main para testar as classes criadas.
 - a) Crie um objeto **camelo** do tipo Mamífero e atribua os seguintes valores para seus atributos:
 - Nome: Camelo
 - Comprimento: 150cm
 - Patas: 4
 - Cor: Amarelo
 - Ambiente: Terra
 - Velocidade: 2.0m/s
 - Alimento: null

Exercício de Herança

4. (cont.)

b) Crie um objeto **tubarao** do tipo Peixe e atribua os seguintes valores para seus atributos

- Nome: Tubarão
- Comprimento: 300cm
- Patas: 0
- Cor: Cinzento
- Ambiente: Mar
- Velocidade: 1.5m/s
- Característica: Barbatanas e cauda

Exercício de Herança

4. (cont.)

- c) Crie um objeto **ursocanada** do tipo Mamifero e atribua os seguintes valores para seus atributos:
 - Nome: Urso-do-canadá
 - Comprimento: 180cm
 - Patas: 4
 - Cor: Vermelho
 - Ambiente: Terra
 - Velocidade: 0.5m/s
 - Alimento: Mel

Exercício de Herança

- 4. (cont.)
 - d) Chame os métodos para imprimir os dados de cada um dos objetos criados.

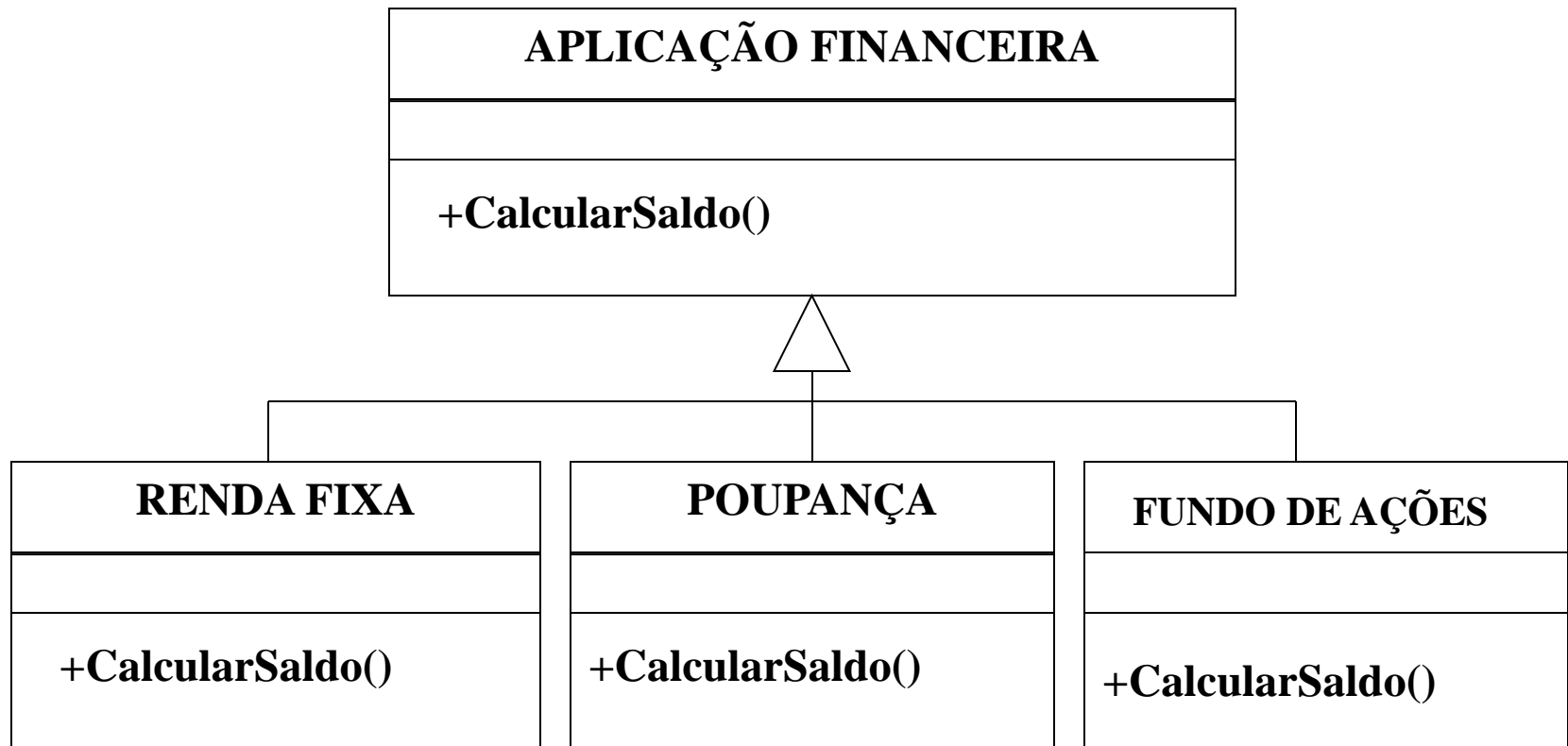
Polimorfismo



Palavra originária do grego “*multas formas*”.

Tais formas se referem aos vários comportamentos que uma mesma operação pode assumir.

Polimorfismo



Polimorfismo

78

```
Class Veiculo{  
  int Velocidade;  
  String Motor="Desligado";  
  public void Ligar() { Motor = "Ligado"; }  
  public void Desligar() { Motor = "Desligado"; }  
  public void Acelera() { Velocidade++; }  
  public void Desacelera() { Velocidade--; }  
}
```

```
Class VeiculoTerrestre extends Veiculo  
{  
  public void Acelera()  
  { Velocidade = Velocidade + 5; }  
  public void Desacelera()  
  { Velocidade = Velocidade - 5; }  
}
```

```
Class VeiculoAquatico extends Veiculo  
{  
  public void Acelera()  
  { Velocidade = Velocidade + 10; }  
  Public void Desacelera()  
  { Velocidade = Velocidade - 10; }  
}
```

Polimorfismo

79

```
public class ExemploDeHeranca{
    public static void main( String Args[] ) {
        VeiculoTerrestre V1;
        VeiculoAquatico V2;
        // Instanciando os OBJETOS
        V1 = new VeiculoTerrestre();
        V2 = new VeiculoAquatico();
        // Utilizando os OBJETOS
        V1.Ligar();
        V2.Ligar();
        System.out.println("Velocidade de V1 = " + String.valueOf(V1.Velocidade));
        System.out.println("Velocidade de V2 = " + String.valueOf(V2.Velocidade));
        V1.Acelera();
        V2.Acelera();
        System.out.println("Velocidade de V1 = " + String.valueOf(V1.Velocidade));
        System.out.println("Velocidade de V2 = " + String.valueOf(V2.Velocidade));
        System.exit(0);
    }
}
```

Abstração

- Abstrair significa transmitir as características relevantes do objeto no mundo real para o sistema;
- Exemplo: um produto pode conter características como
- matéria prima, validade, garantia, preço, fornecedores, quantidade em estoque;
- Na abstração de uma assistência técnica, matéria prima não é relevante;
- Na abstração de uma vendas on-line, preço, garantia e quantidade em estoque são relevantes;

Abstração

- Podemos ter diversos níveis de abstração;
- Devemos trabalhar com abstração do simples para o complexo, do pequeno para o grande, do abstrato para o concreto;
- Exemplo: antes de criar uma classe ContaCorrente, devemos criar um classe Conta com as características comuns para todas as contas;
- Exemplo: antes de criar a classe ClientePessoaFisica, devemos criar uma classe Cliente com as características comuns para todas os clientes;

Classe Abstrata

Uma classe Abstrata é uma classe que:

- ▣ Provê organização;
- ▣ Não possui “instances”;
- ▣ Possui uma ou mais operações(métodos) abstratos.

Uma operação abstrata só determina a existência de um comportamento não definindo uma implementação.

Classe Abstrata

Uma classe abstrata não pode ser instanciada, ou seja, não há objetos que possam ser construídos diretamente de sua definição. Por exemplo, a compilação do seguinte trecho de código.

```
abstract class AbsClass {  
    public static void main(String[] args) {  
        AbsClass obj = new AbsClass();  
    }  
}
```

geraria a seguinte mensagem de erro:

AbsClass.java:3: class AbsClass is an abstract class.

It can't be instantiated.

```
AbsClass obj = new AbsClass();
```

^

1 error

Classe Abstrata

Em geral, classes abstratas definem um conjunto de funcionalidades das quais pelo menos uma está especificada mas não está definida ou seja, contém pelo menos um método abstrato, como em:

```
abstract class AbsClass {  
    public abstract int umMetodo();  
}
```

Classe Abstrata

85

```
public class PessoaFisica extends Pessoa {
    int idade;
    public PessoaFisica(String nome){
        super(nome);
    }
    public static void main(String args[]){
        PessoaFisica pf = new PessoaFisica("Duke");
        pf.setIdade(10);
        System.out.println("idade: " + pf.getIdade());
        System.out.println("nome: " + pf.getNome());
    }
    public String getNome(){
        return nome;
    }
    public int getIdade(){
        return idade;
    }
    public void setIdade(int idade){
        this.idade = idade;
    }
}
```

Classe Abstrata

86

```
abstract class Pessoa{  
    String nome;  
    public Pessoa(String nome){  
        this.nome = nome;  
    }  
    abstract void setIdade(int idade);  
}
```

Interface

- Podemos dizer que interfaces é a forma mais pura de abstração. A interface fornece um contrato entre a classe cliente, que pode ser concreta ou abstrata.
- Este contrato é a garantia que todos os métodos assinados na interface serão implementados na classe cliente.

Interface

- Podemos dizer que interfaces é a forma mais pura de abstração. A interface fornece um contrato entre a classe cliente, que pode ser concreta ou abstrata.
- Este contrato é a garantia que todos os métodos assinados na interface serão implementados na classe cliente.

Interface

89

```
public class Empresa implements Balanco {  
    void calculo(){  
        int valor = 10 + 10;  
    }  
}
```

```
interface Balanco {  
    void calculo();  
}
```