

JAVA Orientado a Objetos

Conheça os fundamentos que embasam a disciplina de orientação a objetos e a sua importância na representação do mundo real

Agenda

- Classes e Objetos
- Abstração
- Encapsulamento
- Herança
- Polimorfismo



DEFININDO CLASSES E OBJETOS

Você ainda lembra o que são as classes e os objetos?



Classes e Objetos

- Como já percebido, a **classe** é uma forma de definir um tipo de dado em uma linguagem O. a Objetos.
 - Ela é formada por dados (definidos através de **atributos**) e comportamentos (definidos pelos **métodos**).
- Depois que uma classe é definida, diferentes objetos podem ser criados a partir dela.
 - Os **objetos** são, então, *instâncias* em tempo de execução de uma classe.
- Assim, a **programação orientada a objetos baseia-se na definição de classes e na criação de objetos** a partir dessas classes, durante a execução do programa.

Como criar um programa orientado a objetos?



Definindo Classes

```
class Carro {  
  
    double peso, tanque;  
    String marca, modelo;  
  
    void Acelerar() {  
        System.out.println("Carro acelerando");  
    }  
}
```

Neste momento é definido o nome da **CLASSE**

Neste ponto são definidos os **ATRIBUTOS** da classe, com seus respectivos tipos de dados

A partir deste ponto são definidos os **MÉTODOS** da classe. Cada método terá a implementação planejada pelo desenvolvedor

Note que todo em todo método é definido o **TIPO DE DADO** do retorno. Como este método não possui retorno, utiliza-se a palavra **VOID**.

Criando o Programa

```
class Programa {  
    public static void main(String[] args) {  
  
        Carro carango = new Carro();  
  
        carango.marca = "Fiat";  
        carango.modelo = "Siena";  
        carango.peso = 2500;  
        carango.tanque = 42.4;  
  
        carango.Acelerar();  
    }  
}
```

Uma das classes do programa será responsável pela instanciação dos objetos e execução do programa, chamada **MAIN** (Principal)

No **JAVA**, este método especial é responsável pela execução do programa

Neste ponto um novo objeto está sendo criado, a partir de uma classe, através do comando **NEW** (novo). Neste momento, o objeto passa a alocar um espaço na memória, como uma variável.

Neste momento os valores dos atributos do objeto criado são informados. Observe a sintaxe hierárquica

O método da classe é evocado, e os comandos internos desse método são executados. A sintaxe também é hierárquica

Exemplo 1:

Classe **Calcular Média**

```
class Media {
```

```
    double nota1, nota2, nota3, nota4;
```

```
    double CalcularMedia() {  
        media = (nota1 + nota2 + nota3 + nota4) / 4;  
        return media;  
    }
```

```
}
```

Neste momento é definido o nome da **CLASSE**

Neste ponto são definidos os **ATRIBUTOS** da classe, com seus respectivos tipos de dados

A partir deste ponto são definidos os **MÉTODOS** da classe. Cada método terá a implementação planejada pelo desenvolvedor

Note que em todo método é definido o **TIPO DE DADO** do retorno

A palavra chave **RETURN** define qual será o retorno do método. Este retorno deve ser do tipo de dado definido na declaração

Exemplo 1:

Programa **Calcular**

```
class Programa {  
    public static void main(String[] args) {  
  
        Media calculo = new Media();  
  
        calculo.nota1 = 10;  
        calculo.nota2 = 5;  
        calculo.nota3 = 8;  
  
        System.out.println(calculo.CalculaMedia());  
    }  
}
```

Uma das classes do programa será responsável pela instanciação dos objetos e execução do programa, chamada **MAIN** (Principal)

No **JAVA**, esta método especial é responsável pela execução do programa

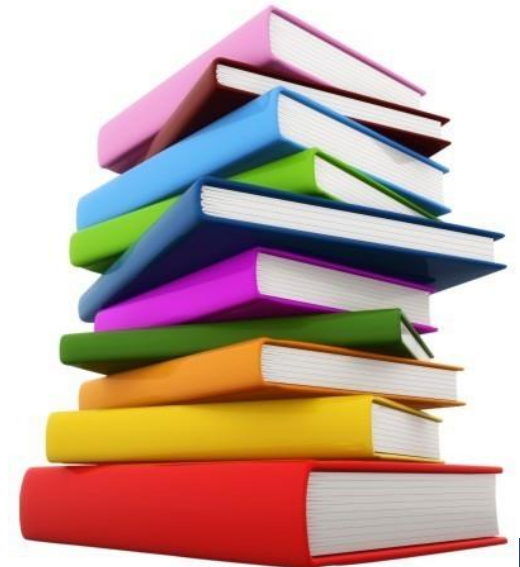
Neste ponto um novo objeto está sendo criado, a partir de uma classe, através do comando **NEW** (novo). Neste momento, o objeto passa a alocar um espaço na memória, como uma variável.

Neste momento os valores dos atributos do objeto criado são informados. Observe a sintaxe hierárquica

O método definido na classe é chamado, e o resultado é retornado para o usuário. A sintaxe também é hierárquica

Exercícios

- **Uma biblioteca** possui um acervo de livros, descritos pelo seu **título**, **editora** e **autor**. O sistema permite que cada livro **imprima** na tela os seus dados. Desenvolva um programa orientado a objetos em *Java* que implemente esse sistema.



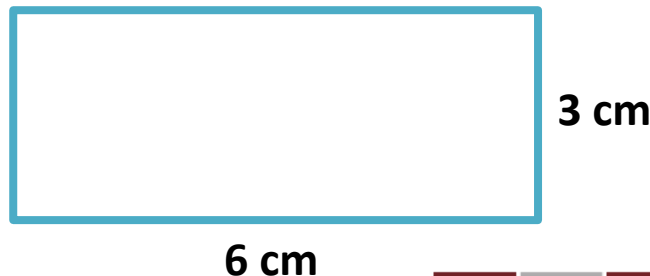
Exercícios

- Considere **um sistema acadêmico**, onde um aluno é representado pela sua matrícula, nome e idade. É possível, para cada aluno, imprimir na tela os seus dados. Desenvolva um programa orientado a objetos em *Java* que implemente esse sistema.



Exercícios

- Um retângulo pode ser representado pela sua **base** e **altura**. A partir desses dados, é possível extrair informações sobre a **área do retângulo**, o **perímetro**, e se o retângulo **é ou não um quadrado**. Desenvolva um programa orientado a objetos em *Java* que represente um retângulo e manipule-o.



Exercício (desafio)



- Crie um **sistema de lanchonete**, onde cada **cliente** possui um **total a pagar**, **hora de chegada**, **hora de saída** e os seguintes métodos:
 - Sempre que o cliente **realiza um pedido**, um valor é passado (por parâmetro) e é adicionado no total a pagar.
 - O cliente pode **solicitar a conta**, a qual exibirá na tela o total a pagar.
 - O cliente pode solicitar **tempo de atendimento**, onde é exibido na tela o cálculo da hora de saída menos a hora de chegada.

Entenda o conceito de abstração e sua importância no paradigma da orientação a objetos

ABSTRAÇÃO

**Pra você, o que é uma
ABSTRAÇÃO?**



Abstração

- **Abstração** é o processo mental responsável pela identificação dos aspectos mais importantes que caracterizam um grupo, desprezando propriedades menos relevantes.
 - Este processo permite classificar os diversos objetos existentes no mundo real, independente das suas complexidades;
 - Exemplo: Uma *Ferrari* é igual a um *Hot Wheels*?



Abstração (cont)



Abstração (cont)



Como utilizar Abstração na programação?



Exemplo de Abstração

- **PROBLEMA:**
 - Criar um programa para cadastro e listagem de empresas.
- **ABSTRAÇÃO:**
 - O que vou precisar ter/fazer nesse programa?
 - Quais as classes preciso definir? Como elas irão se relacionar?
 - Como posso representar as empresas? Quais atributos e métodos cada empresa precisa ter?
 - Como posso representar o cadastro? E a listagem?

Exemplo de Abstração (cont.)

```
public class Empresa {
```

```
    String nome;
```

```
    String cnpj;
```

```
    String endereco;
```

```
    int anoFundacao;
```

Você está usando **ABSTRAÇÃO** quando:

Escolhe as informações relevantes e necessárias para descrever a classe

```
public void listarDados() {
```

```
    System.out.println("Nome: " + nome);
```

```
    System.out.println("CNPJ: "
```

```
    System.out.println("Endereç
```

```
    System.out.println("Ano de
```

```
        anoFun
```

Define os métodos necessários para a utilização da classe no programa, de acordo com o problema proposto;

Exemplo de Abstração (cont.)

```
public static void main(String[] args) {  
    Empresa vale = new Empresa();  
  
    vale.nome = "Vale";  
    vale.cnpj = "111134444\113";  
    vale.endereco = "Sao Luis";  
    vale.anoFundacao = 1990;  
  
    vale.listarDados();  
}
```

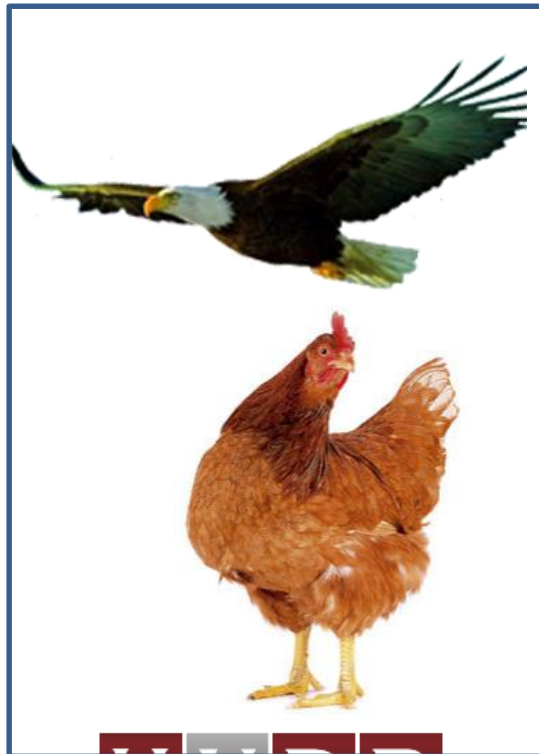
Exercícios

- Em uma revendedora de veículos, é possível encontrar diversos tipos de transportes, como os da figura abaixo. Crie uma classe da qual possam ser criados todos os objetos da figura abaixo, com o máximo de característica que você conseguir.



Exercícios

- Utilizando o princípio da abstração, crie uma classe para cada grupo abaixo, das quais possam ser criados todos os objetos de cada grupo.



Exercícios

Com apenas 19 anos de idade, Paulo é o candidato a vereador mais novo do município de Barreto e seu número de campanha (1919) faz referência a isso. Por outro lado, seu pai Zeca (48 anos) é candidato a prefeito pela segunda vez e possui número de campanha 40 por ser a idade que ele tinha em seu primeiro mandato.

- A partir dessas informações e utilizando o princípio da abstração, crie uma classe Candidato da qual seja possível criar os objetos Paulo e Zeca com o maior número de características possíveis.

Exercícios



Beto tem 29 anos e trabalha como eletricista em uma empresa desde 2005. Ele recebe um salário de R\$ 1.900 todo dia 05 de cada mês e, entre suas obrigações, Beto é responsável por registrar ponto, atender ordens de serviço e gerar relatórios.

- Com base nessas informações e utilizando o principio da abstração, crie uma classe **Empregado** da qual um objeto Beto possa ser instanciado.
- Crie um objeto Maria utilizando a mesma classe **Empregado** e atribua valores apropriados a ela.



Exercícios (**Desafio**)

- A renda obtida em um jogo de futebol é baseada no número de ingressos vendidos. Ao final do jogo, o time mandante (dono da casa) recebe 70% da renda e o time visitante 30%.

Sabendo disso, crie um programa orientado a objeto que permita realizar a contabilidade de vários jogos de futebol.





Conheça a propriedade da orientação a objetos responsável pela ocultação de código

ENCAPSULAMENTO



O que significa encapsular?



Encapsulamento

- Propriedade da orientação a objeto que restringe o acesso ao comportamento interno (atributos e métodos) de uma classe.
 - Com relação aos **atributos**, restringe o acesso deles por outros objetos.
 - Com relação aos **métodos**, oculta seu funcionamento interno tornando-o uma caixa preta para as demais classes desse programa e de programas externos.
 - Torna cada classe encapsulada independente das demais;
 - Esta propriedade possibilita o uso do software por outros programas sem perder/ compartilhar a regra do negócio (código interno).

Encapsulamento (cont.)



O controle é sua interface de acesso aos métodos encapsulados do jogo.

■ Como utilizar Encapsulamento na programação?



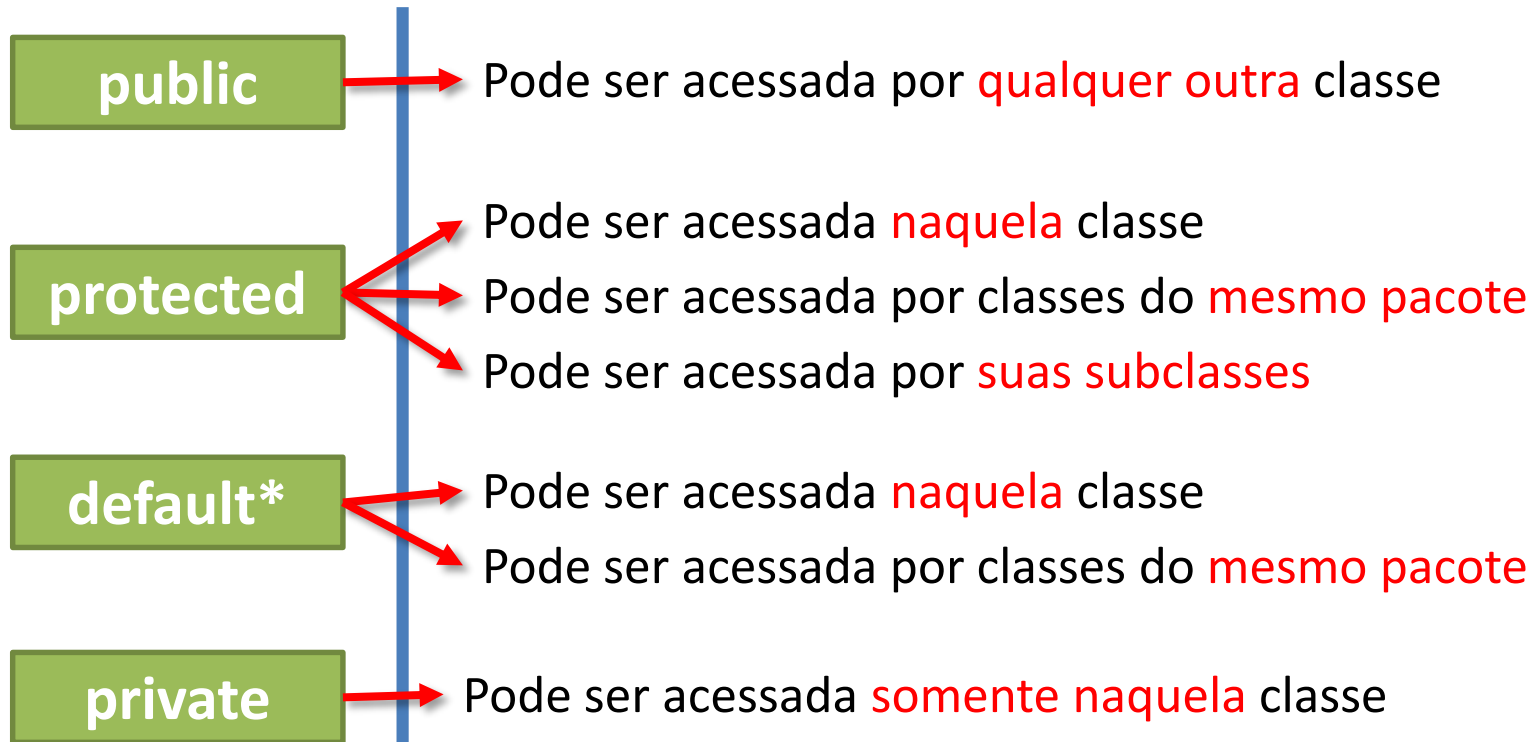
Encapsulamento (cont.)

- O **encapsulamento** é obtido através do bloqueio do acesso direto aos atributos e métodos particulares de uma classe.
 - Para isso, definimos a **visibilidade** desses atributos e métodos, determinando seus níveis de acesso.
- Ao bloquear (**privar**) o acesso a atributos e métodos exclusivos de uma classe, outros objetos **nunca os manipularão diretamente**.
- Já os métodos/atributos **públicos** estarão visíveis para todos os objetos do programa, possibilitando a interação entre eles.

Visibilidade

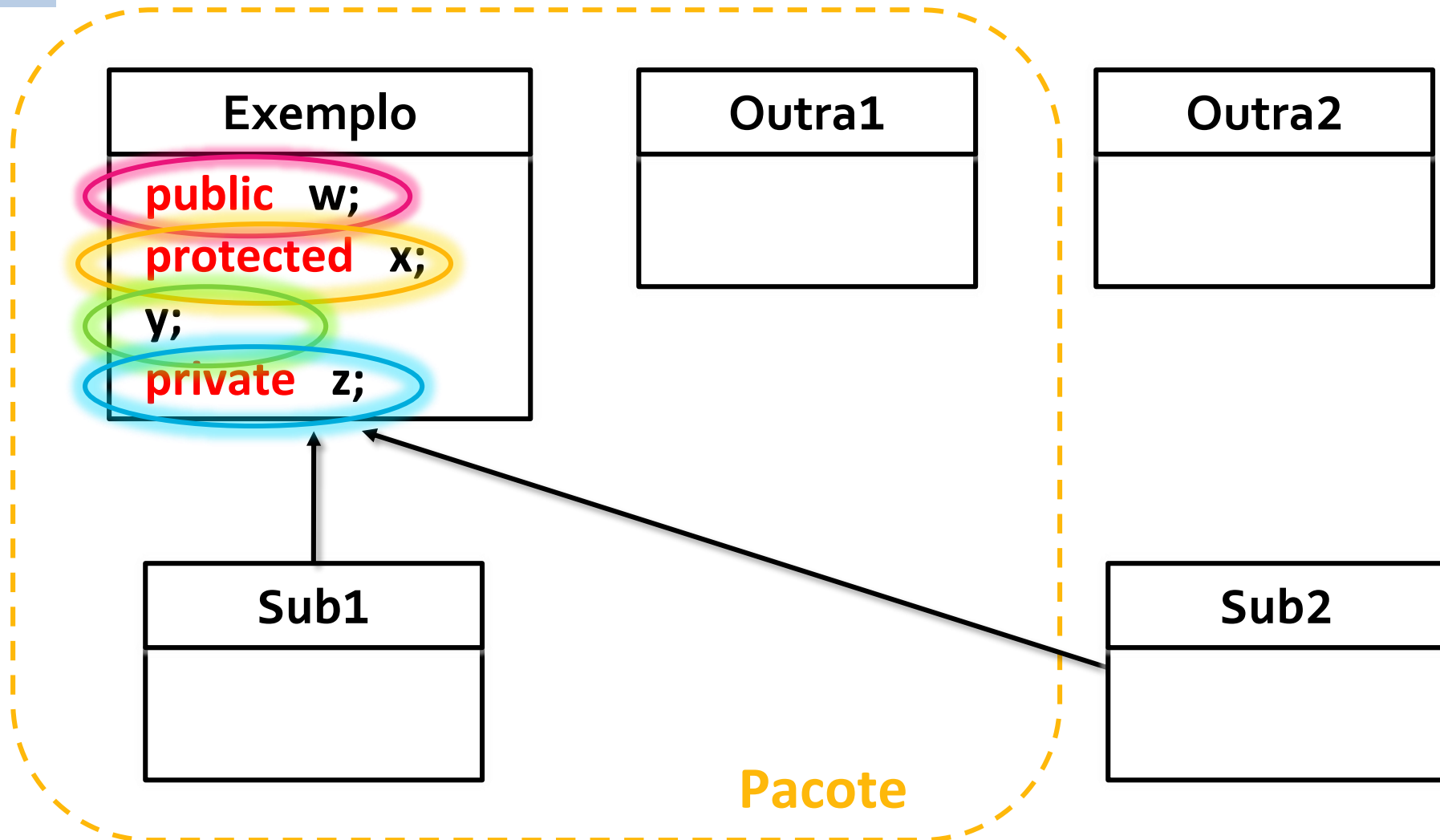
- São quatro níveis de visibilidade:

Menos Restrição



Mais Restrição

Visibilidade (cont.)



Visibilidade (cont.)

```
class Visibilidade {
```

```
    public int numero1;
```

```
    private int numero2;
```

```
    public void metodoPublico() {
```

```
        System.out.println("Método Público!");
```

```
    }
```

```
    private void metodoPrivado() {
```

```
        System.out.println("Método Privado!");
```

```
    }
```

```
}
```

Observe as diferentes visibilidades dos atributos e métodos desta classe.

Visibilidade (cont.)

```
class Programa {
```

```
    public static void main(String[] args)
```

```
    {
        Visibilidade v = new Visibilidade()
```

```
        {
            v.numero1 = 10;
```

```
            v.numero2 = 20;
```

```
            v.metodoPublico();
```

```
            v.metodoPrivado();
```

```
        }
```

```
    }
```

Este atributo estará visível para o programa principal, pois foi declarada como **PUBLIC**

Este atributo **NÃO** estará visível para o programa principal, pois foi declarada como **PRIVATE**

Este método estará visível para o programa principal.

Este método **NÃO** estará visível para o programa principal.

Como acessar e manipular os atributos *Private*?



Controladores de Acesso

- Para garantir o acesso e a manipulação de dados armazenados em classes com atributos e métodos privados, definem-se métodos responsável por possibilitar a troca de mensagens entre objetos encapsulados em um programa.
- São dois tipos de métodos:
 - **Modificadores** (*setters*)
 - **Recuperadores** (*getters*)



Modificadores (Set)

- São métodos que permitem a modificação dos valores armazenados nos atributos privados de um objeto.
 - Dessa forma, objetos externos podem alterar **indiretamente** o valor de um ou mais atributos;
- Como serão acessados por outras classes, a visibilidade dos modificadores deve ser pública.
- Uma boa prática é a utilização do prefixo **SET** no nome dos métodos modificadores (*definir* em inglês).
 - O uso do prefixo SET facilita a identificação e compreensão no código-fonte de uma classe;
 - Também chamados de *setters*;
 - Exemplo: **void** setMatricula(**int** numero) { ... }

Modificadores (cont.)

```
class Cliente {
```

```
    private int codigo;  
    private String nome;  
    private byte idade;
```

```
    public void setCodigo(int a)  
    {  
        this.codigo = a;  
    }
```

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

```
    public void setIdade(int c)  
    {  
        this.idade = (byte) c;  
    }
```

```
}
```

Veja que os atributos da classe são **PRIVADOS**, não sendo portanto visíveis a outras classes. Além disso, não há **Construtores** personalizados para essa classe.

Entretanto, observe que os métodos **MODIFICADORES** permitem definir valores aos atributos de forma indireta

A palavra **THIS** permite referenciar o objeto a quem o método pertence. Em outras palavras, os atributos informados pertencem ao **MESMO** objeto que está executando a ação.

Modificadores (cont.)

```
class Programa {  
  
    public static void main(String[] args) {  
  
        Cliente x = new Cliente();  
  
        x.codigo = 77;  
  
        x.setCodigo(77);  
        x.setNome("Sicrano");  
        x.setIdade(45);  
  
    }  
}
```

Como o atributo é privado, ele não pode ser utilizado diretamente.

O modo correto para definir valores em atributos privados é através dos métodos modificadores.

Recuperadores (Get)

- São método que permitem a visualização dos valores armazenados nos atributos privados de um objeto.
 - Dessa forma, objetos externos podem acessar **indiretamente** o valor de um atributo;
- Como serão acessados por outras classes, a visibilidade dos métodos de acesso deve ser pública.
- Uma boa prática é a utilização do prefixo **GET** no nome dos métodos de acesso (*recuperar* em inglês).
 - Facilitando a sua identificação e compreensão no código-fonte de uma classe;
 - Também chamados de *getters*;
 - Exemplo: **public int** getMatricula() { **return** matricula; }

Recuperadores (cont.)

```
class Aluno {  
    private int matricula;  
    private String nome;  
    public void setMatricula(int a) {  
        this.matricula = a;  
    }  
    public void setNome(String b) {  
        this.nome = b;  
    }  
    public int getMatricula() {  
        return this.matricula;  
    }  
    public String getNome() {  
        return this.nome;  
    }  
}}
```

Veja que os atributos da classe são **PRIVADOS**, não sendo portanto visíveis a outras classes.


Observe a utilização dos métodos **MODIFICADORES**.

Neste ponto, os **RECUPERADORES** são definidos. Repare que cada método possui um tipo de retorno definido, que é o mesmo do atributo retornado pelo método.

Recuperadores (cont.)

```
class Programa {  
  
    public static void main(String[] args) {  
  
        Aluno a1 = new Aluno();  
  
        a1.setMatricula(1234);  
        a1.setNome("Delano");  
  
        System.out.println("Matrícula: " + a1.getMatricula());  
        System.out.println("Nome: " + a1.getNome());  
    }  
}
```

Neste ponto, os métodos de acesso são chamados, permitindo assim o retorno dos valores dos atributos privados.



Exercício

Sistema Bancário



- Criar uma classe **Conta** com atributos privados **nome** e **saldo**;
- Criar métodos modificador (SET) e recuperador (GET) para os atributos nome e saldo;
- Criar uma classe **CaixaEletronico** que utilize os métodos de um objeto da classe Conta da seguinte forma:
 - Defina valores para os atributos saldo e nome através dos métodos modificadores (SETTERS)
 - Imprima na tela o saldo final do usuário utilizando o método recuperador (GET) para acessá-lo.



Exercício

Cadastro de Funcionário

- Em uma empresa é necessário manter o registro dos principais dados dos funcionários, pois em algumas situações é necessário consultar quanto o funcionário ganha, o seu telefone atual ou o seu endereço, por exemplo. Contudo, algumas dessas informações são privadas e não podem ter seu acesso público. Assim, construa um sistema que seja capaz de armazenar informações sobre um funcionário, mas que as alterações e consultas a esses dados sejam feitas de forma indireta, ou seja, sem manipular o atributo privado diretamente.

Exercício

- Em um sistema para calcular a média dos alunos, é necessário três atributos privados (**nota1**, **nota2**, **nota3**). E por serem privados, o acesso a essas notas deve ser feito através de métodos modificadores e recuperadores. Assim, construa uma classe para cadastro das notas de um aluno e realize o cálculo da média com base no preenchimento (set) e na leitura (get) dessas notas.

■ Como passar valores iniciais para os atributos de um objeto?



Construtores

- São chamadas especiais utilizados na criação de TODOS os objetos.
- Eles criam (instanciam) o objeto, ou seja, armazenam e inicializam os valores de seus atributos na memória do sistema computacional.
- Ao declarar o método construtor, ele deve ter o mesmo nome da classe ao qual pertence e não possui retorno;
 - Exemplo: Se a classe chama-se **Professor**, o construtor terá a sintaxe **public Professor()**.
 - O resultado desse método é um objeto instanciado da classe;

Construtores (cont.)

```
class Funcionario {
```

```
    private int codigo;  
    private String nome;  
    private int idade;
```

```
    Funcionario(int a, String b, int c) {  
        this.codigo = a;  
        this.nome = b;  
        this.idade = c;  
    }  
}
```

Observe que o **CONSTRUTOR** não possui um tipo de dado de retorno, e o nome do método é o mesmo da classe do construtor.

Assim como nos métodos, construtores suportam parâmetros de entrada.

A palavra **THIS** permite referenciar o objeto a quem o método pertence. Em outras palavras, os atributos informados pertencem ao **MESMO** objeto que está executando a ação.

Construtores (cont.)

```
class Programa {
```

```
    public static void main(String[] args)
```

Ao instanciar um objeto através do construtor, os valores dos atributos são passados como parâmetros.

```
        Funcionario f = new Funcionario(1, "Fulano", 25);
```

```
        System.out.println("Código: " + f.codigo);
```

```
        System.out.println("Nome: " + f.nome);
```

```
        System.out.println("Idade: " + f.idade);
```

```
    }
```

Toda classe é obrigada a ter um Construtor?



Construtores (cont.)

- A inclusão de um construtor nas classes NÃO é obrigatória.
 - Um construtor padrão é gerado automaticamente quando não há construtor definido;
 - Ao definir um construtor para uma classe, o construtor padrão deixa de existir.
- Embora não obrigatório, recomenda-se a criação de um construtor, especialmente quando:
 - Houver atributos privados que necessitam de valores iniciais;
 - A classe terá subclasses com métodos construtores (cenas dos próximos capítulos);
 - A classe será herdada por outras classes (cenas dos próximos capítulos, **TAMBÉM!**);

Construtores *versus* Métodos

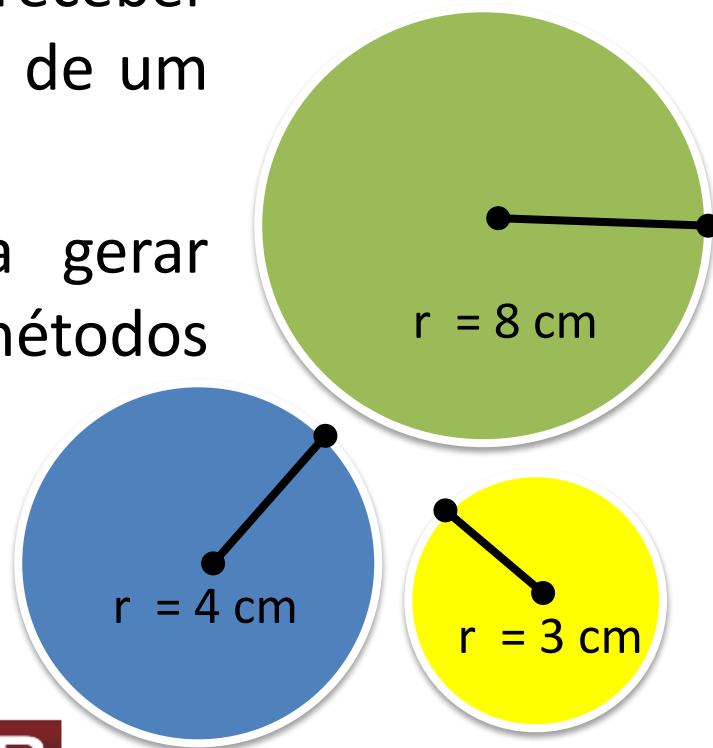
- Apesar de diversas semelhanças estruturais e de utilização, *Construtores* e *Métodos* são membros distintos de uma classe.
 - **Construtores** são utilizados para criar e inicializar objetos que ainda não existem, enquanto **métodos** realizam operações em objetos que já foram criados;
 - **Construtores** não são chamados diretamente pelo programa principal, mas de forma implícita através do comando **new**. **Métodos** são chamados diretamente, a partir do objeto que possui a implementação dessas operações e utilizando a notação hierárquica.

Exercício

1. Crie uma classe **Círculo** com o atributo privado **raio**.
2. Crie métodos públicos para calcular a **área** e o **perímetro** do círculo utilizando as fórmulas abaixo.
3. Crie um método construtor para receber o valor do raio durante a criação de um objeto Círculo.
4. Crie uma classe principal para gerar objetos círculos e invocar os métodos públicos deles.

$$\text{área} = r^2 * \pi$$

$$\text{perímetro} = 2 * \pi * r$$



Exercício

Controle de Acesso



- Crie uma classe **Seguranca** que possui os atributos privados: **usuarioPadrao**, **senhaPadrao**;
- Crie um método construtor que receberá valores e os atribua a **usuarioPadrao** e **senhaPadrao** de modo que estejam definidos desde o momento da criação do objeto;
- Crie um método **autenticacao** que recebe dois parâmetros de entrada e os compara com o **usuarioPadrao** e **senhaPadrao**. Caso ambos sejam iguais, imprima na tela “Autenticação realizada com sucesso”; Caso contrário, notifique falha;
- Crie uma classe principal que instancie um objeto da classe **Seguranca**, pergunte pro usuário um nome e senha e chame o método **autenticacao** para verificar se ele pode acessar o sistema ou não.

Exercício

- Crie uma classe **Loteria** que possui três atributos privados do tipo inteiro (n1, n2, n3);
- Crie um método construtor que receba três parâmetros de entrada, um para cada atributo da classe.
- Crie um método **checarSequencia** que receba três valores (a1, a2, a3) como parâmetros de entrada e verifique se a sequência (a1, a2, a3) é igual a (n1, n2, n3). Caso seja, informe ao usuário que ele ganhou na loteria.
- Crie um programa principal para gerar um objeto Loteria e depois pergunte ao usuário os número da sequência (a1, a2, a3) e verifique se ele ganhou.

Exercício:

Que número estou pensando

Crie um programa onde um objeto é instanciado passando um valor inteiro qualquer (número secreto) através do seu método construtor. Em seguida, peça ao usuário que tente adivinhar o número secreto.

Ao receber o palpite do usuário, utilize um método do objeto para comparar o palpite com o número secreto. Caso esteja errado, o usuário deve tentar novamente.

O programa encerra apenas quando o usuário acertar o número secreto.





Veja a propriedade da orientação a objetos responsável pelo reaproveitamento de código

HERANÇA



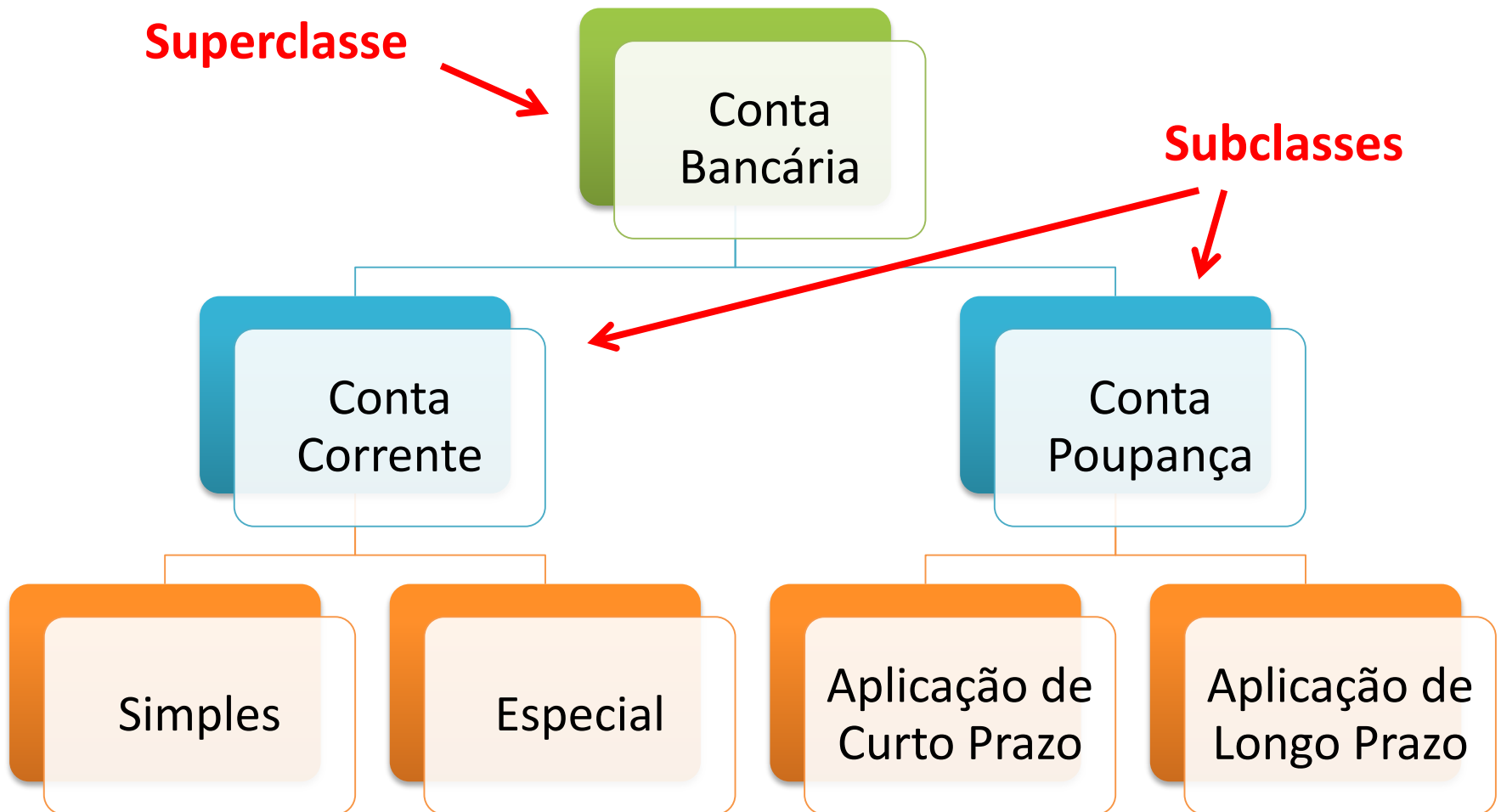
O que te faz ser filho do teu pai?



Herança

- Propriedade da orientação a objeto que permite a transmissão das características de uma classe (atributos e métodos) para outra classe relacionada hierarquicamente.
 - Esta propriedade permite a criação de novas classes a partir de uma classe principal;
 - A classe que possui suas características herdadas é chamada de *classe pai* ou **superclasse**;
 - A classe que herda características de outra classe é chamada de *classe filho* ou **subclasse**.
- A herança permite a **reutilização** de propriedades já existentes em uma classe na definição de novas.

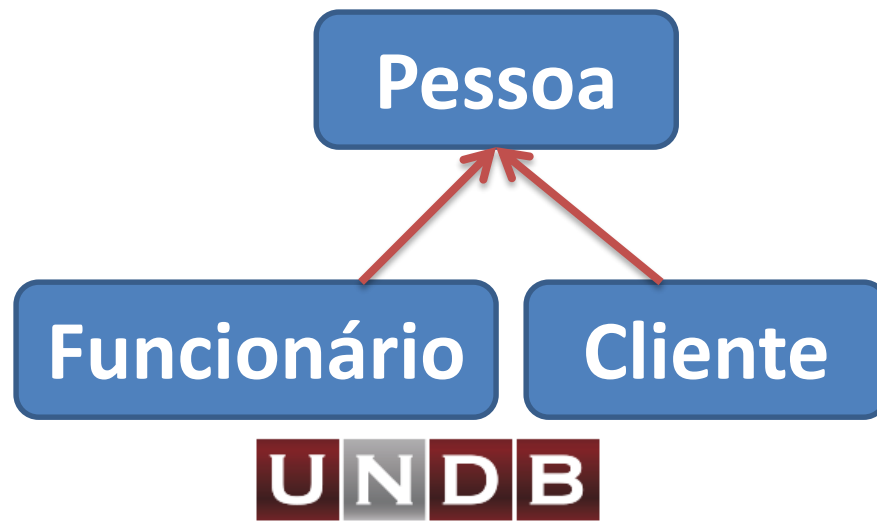
Herança (cont.)



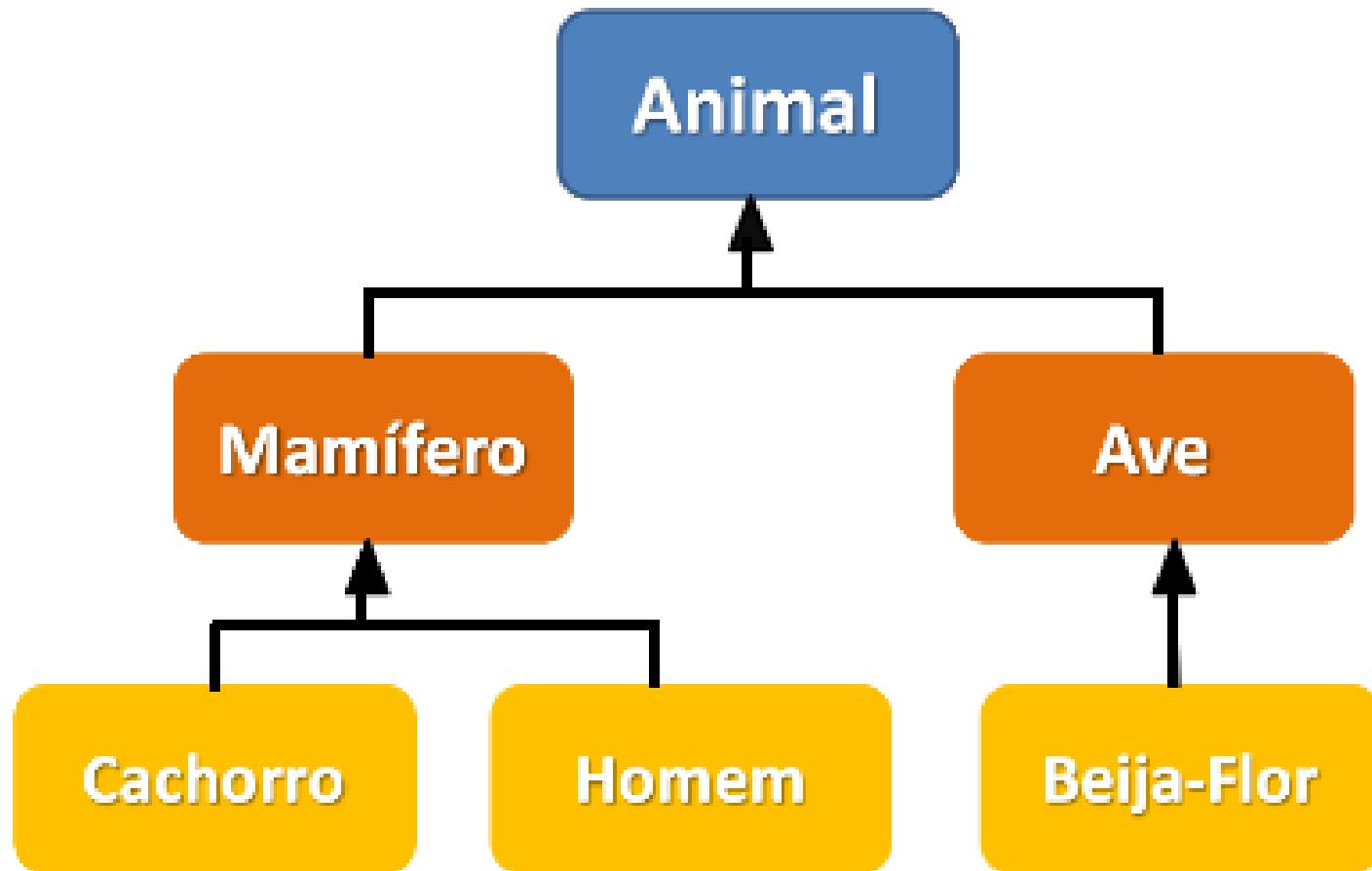
Se a classe pai for alterada, automaticamente os filhos também serão.

Herança (cont.)

- Dois processos relacionados a herança:
 - **Especialização**: construção de novas classes a partir de uma mais geral (mais abstrata).
 - **Generalização**: construção de uma nova classe a partir de classes mais especializadas (menos abstrato).
- Especialização e generalização são processos inversos. Portanto, o resultado final de ambos é o mesmo;



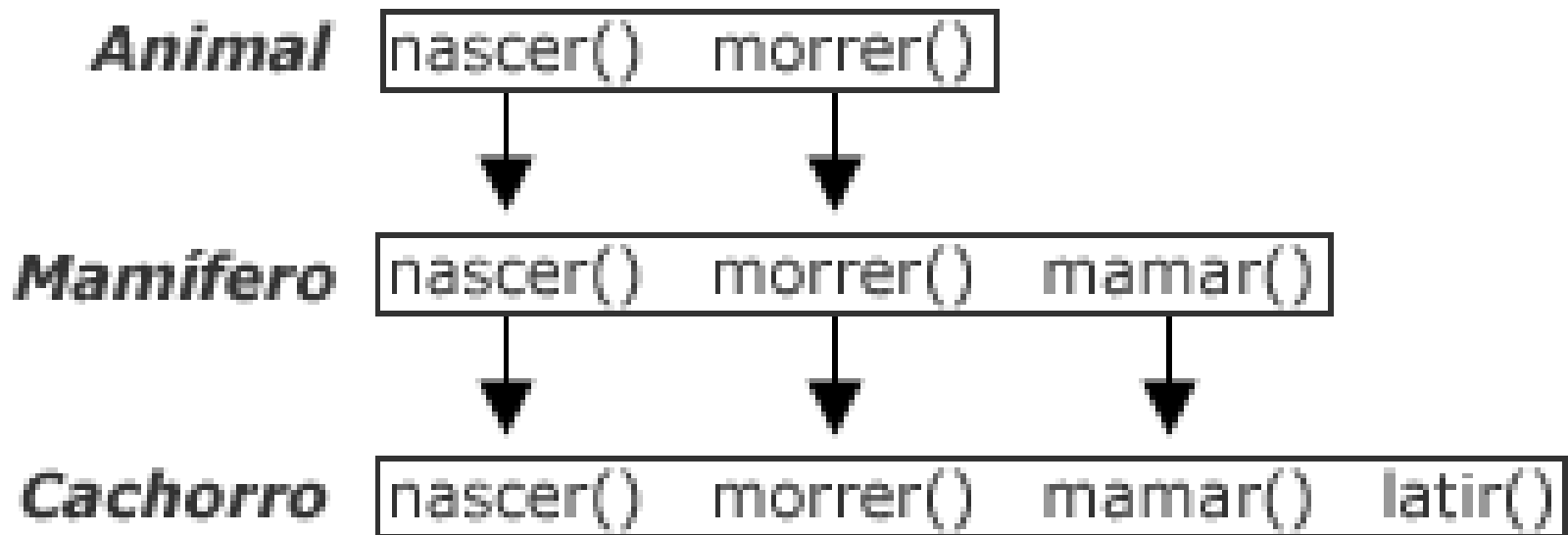
Herança (cont.)



Para saber se estamos aplicando a herança corretamente, realiza-se o teste **"É UM"**.

Herança (cont.)

Lembre-se que, na herança, os métodos da classe pai também são herdados.



Como utilizar Herança na programação?



Herança (cont.)

- Na programação orientada a objetos, a herança é implementada através da referência à classe pai durante a definição de uma subclasse.
 - Esta referência **estende** a classe principal, formando o vínculo hierárquico;
- Através da herança, objetos de classes mais especializadas copiam as propriedades (atributos e/ou métodos) de classe mais genéricas com as quais estão relacionadas.
- As propriedades a serem herdadas pelas subclasses dependem das visibilidades definidas na classe pai.

Herança (cont.)

```
class Cliente {
```

Essa classe Cliente será nossa classe pai.

```
public int codigo;
```

```
public String nome;
```

Observe os atributos definidos na classe pai.

```
public void escreverCliente() {
```

```
    System.out.println("Código: " + this.codigo);
```

```
    System.out.println("Nome: " + this.nome);
```

```
}
```

```
}
```

Veja que esta classe também possui um método definido.

Herança (cont.)

```
class ClientePF extends Cliente {
```

```
    public String cpf;
```

```
    public void escreverCPF() {
```

```
        System.out.println("CPF: " + this.cpf);
```

```
    }
```

```
}
```

Observe que esta nova classe faz uma referência a outra classe criada anteriormente, através do termo **EXTENDS**. Neste momento, a herança foi estabelecida.

Observe o atributo especializado definido na subclasse.

Veja que esta subclasse também possui um método.

Herança (cont.)

```
class Programa {
```

```
    public static void main(String[] args) {
```

```
        Cliente x = new Cliente();
```

```
        x.codigo = 10;
```

```
        x.nome = "Fulano";
```

```
        x.escreverCliente();
```

Nesta primeira parte do programa, a criação do objeto *x* é apenas para provar que a classe *Cliente* existe.

Observe os atributos e métodos sendo chamados normalmente.

...

Herança (cont.)

Veja que agora um novo objeto é criado a partir da subclasse ClientePF.

...
`ClientePF z = new ClientePF();`

`z.codigo = 20;`

`z.nome = "Sicrano";`

`z.cpf = "123.456.789-00";`

`z.escreverCliente();`

`z.escreverCPF();`

`}`

`}`

Graças a herança, os atributos da classe pai também estão disponíveis para a subclasse.

O atributo especializado também está disponível, mas só para esta classe.

A mesma regra se aplica aos métodos.

Herança (cont.)

- Portanto, a herança constitui um poderoso mecanismo para o reaproveitamento de código nos programas orientados a objetos.
 - O objeto instanciado a partir de uma subclasse possui também os atributos e métodos da classe pai, sem a necessidade de cópia dessas propriedades na subclasse;
- A herança também facilita a manutenção do código.
 - Alterações realizadas nas propriedades de uma classe pai são, graças à herança, aplicadas automaticamente nas subclasses.
- É importante entender que, na herança, não há replicação (cópia) das propriedades da classe pai, mas sim a definição de uma referência (apontamento) para a classe principal.

■ **É possível uma classe possuir mais de uma subclasse?**



Herança com Várias Subclasses

```
class Venda {
```

```
    public int NF;
```

```
    public String cliente;
```

```
    public double valor;
```

```
    public void escreverVenda() {
```

```
        System.out.println("Nota Fiscal: " + this.NF);
```

```
        System.out.println("Nome: " + this.cliente);
```

```
        System.out.println("Valor: " + this.valor);
```

```
    }
```

```
}
```

Observe os atributos definidos na classe principal.

Veja que esta classe também possui um método definido.

Herança com Várias Subclasses

```
class VendaBalcao extends Venda {
```

Observe a herança na criação desta nova classe.

```
public String vendedor;
```

Confira o atributo especializado definido na subclasse.

```
public void escreverVendaBalcao() {
```

```
    escreverVenda();
```

```
    System.out.println("Vendedor: " +
```

```
this.vendedor);
```

```
}
```

```
}
```

Um fato interessante está representado neste método especializado. Observe que este método executa o método definido na classe pai.

Herança com Várias Subclasses

```
class VendaOnline extends Venda {  
  
    public String IP;  
  
    public void escreverVendaOnline() {  
        escreverVenda();  
        System.out.println("IP: " + this.IP);  
    }  
}
```

Repare que outra classe está também fazendo referência a classe pai.

Herança com Várias Subclasses

```
class Programa {
```

```
public static void main(String[] args) {
```

```
VendaBalcao a = new VendaBalcao();
```

```
a.NF = 1234;
```

```
a.cliente = "Fulano";
```

```
a.valor = 523.77;
```

```
a.vendedor = "Chico Bento";
```

```
a.escreverVendaBalcao();
```

...

Veja que um objeto da primeira subclasse é instanciado.

Observe os atributos sendo chamados normalmente.

Repare que apenas um método foi chamado, trazendo todas as informações do objeto.

Herança com Várias Subclasses

...

```
VendaOnline b = new VendaOnline();
```

Veja que agora um objeto da segunda subclasse é instanciado.

```
b.NF = 4321;
```

```
b.cliente = "Sicrano";
```

```
b.valor = 180.50;
```

```
b.IP = "192.168.0.100";
```

Observe os atributos sendo chamados normalmente.

```
b.escreverVendaOnline();
```

```
}
```

```
}
```

Repare que apenas um método foi chamado, trazendo todas as informações do objeto.

■ **Será que eu posso criar subclasses
a partir de outras subclasses?**



Herança com Vários Níveis

```
class Conta {
```

```
    public int agencia;  
    public int numero;
```

```
}
```

Observe que esta subclasse herda as propriedades da classe pai...

```
class ContaCorrente extends Conta {
```

```
    public boolean cartaocredito;
```

```
}
```

```
class ContaCorrenteEsp extends ContaCorrente {
```

```
    public double limite;
```

```
}
```

... e que esta subclasse herda as propriedades da subclasse definida anteriormente.

Herança com Vários Níveis

Veja que um objeto da subclasse mais especializada é instanciado.

```
class Programa {  
    public static void main(String[] args) {  
        ContaCorrenteEsp a = new ContaCorrenteEsp();  
        a.agencia = 1234;  
        a.numero = 98765;  
        a.cartaocredito = true;  
        a.limite = 5000.00;  
        System.out.println(a.agencia + " | " + a.numero + " | " + a.cartaocredito + " | " + a.limite);  
    }  
}
```

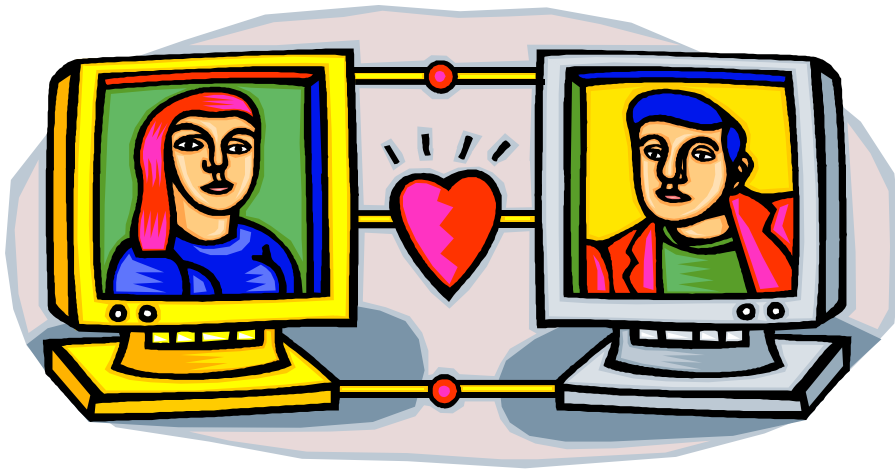
Observe os atributos herdados da primeira classe (mais abstrata).

Observe o atributo herdado da primeira subclasse (menos abstrata).

Observe o atributo especializado da subclasse instanciada.

Herança (cont.)

- Portanto, uma classe pode ser pai de várias subclasses.
 - Como resultado suas propriedades são herdadas para todos os filhos dessa classe.

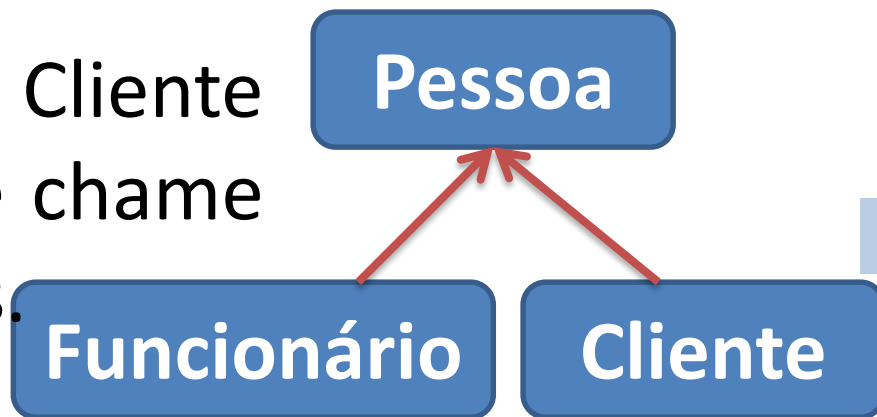


Além disso, é possível definir vários níveis de subclasses.

- Permitindo definir uma hierarquia em vários níveis.

Exercício

- Crie uma classe **Pessoa** e defina alguns atributos e métodos genéricos que toda “pessoa” tem.
- Crie duas outras classes chamadas **Funcionário** e **Cliente** (que herdam características de Pessoa) e defina um atributo e um método em cada uma delas.
- Crie um objeto da classe Cliente e da classe Funcionario e chame seus respectivos métodos.



Exercício

Zoológico

- Um **animal** contém um nome, número de patas, uma cor e um habitat.
- Um **peixe** é um **animal** e tem como característica duas barbatanas;
- Um **mamífero** é um **animal** e é capaz de mamar;
- Um **urso** é um **mamífero** e seu alimento preferido é o mel.

Abstraia as informações e codifique as classes animal, peixe, mamífero e urso e estabeleça as relações hierárquicas entre elas.

- Instancie objetos das classes acima e crie os seguintes animais: **camelo**, **tubarao**, **urso_panda**.

Modificadores de Classe

- Por padrão, toda classe pode ser instanciada como objeto e ter suas propriedades herdadas.
- Entretanto, em alguns cenários, pode-se limitar o comportamento de uma classe através da utilização de ***modificadores de classe***.
 - Influenciam o comportamento da classe em um programa.
- Dois modificadores disponíveis no *Java* para o controle de herança:
 - ***ABSTRACT***: Indica que a classe não poderá ser instanciada;
 - ***FINAL***: Indica que a classe não poderá ser estendida.

Modificadores de Classe (cont.)

```
abstract class Produto {  
    protected int codigo;  
    protected String nome;  
    protected double valor;
```

Observe que a classe está com o modificador **ABSTRACT**, indicando que esta classe não pode ser instanciada.

```
}  
  
final class ProdutoVenda extends Produto {  
    protected String dataretirada;  
    ProdutoVenda(int a, String b, double c) {  
        this.codigo = a;  
        this.nome = b;  
        this.valor = c;
```

Observe que a classe está com o modificador **FINAL**, indicando que esta classe não pode ser estendida.

Modificadores de Classe (cont.)

```
class ProdutoExcluido extends ProdutoVenda {  
    protected String dataexclusao;  
}  
  
class Programa {  
    public static void main(String[] args) {  
        Produto x = new Produto();  
        ProdutoVenda y = new ProdutoVenda(1,  
"Teclado", 15.00);  
        System.out.println(y.codigo + " | " +  
y.nome + " | " + y.valor);  
    }  
}
```

Não será possível criar esta classe, uma vez que a classe pai não pode ser estendida.

Não será possível criar este objeto, uma vez que a classe não pode ser instanciada.

**Mas qual a finalidade de uma classe
que não pode ser instanciada?**



Classes Abstratas

- Classes abstratas apenas idealizam um tipo, definem apenas um rascunho.
- Usamos a palavra chave **abstract** para impedir que uma classe possa ser instanciada.
- Não poder instanciar um objeto de uma classe pode ser de grande valia, dando mais consistência ao sistema.
- São utilizadas para polimorfismo e herança dos atributos e métodos, que são recursos muito poderosos.

Acesso a Classe Pai

- Em alguns casos, existe a necessidade de acessar, a partir de uma subclasse, propriedades (atributos ou métodos) definidos na classe pai.
 - A palavra reservada **SUPER** permite acesso às propriedades da classe pai;
 - Permite o reaproveitamento de código, uma vez que é possível utilizar atributos e métodos já existentes;



Acesso a Classe Pai (cont.)

```
abstract class Aluno {
```

```
    private int codigo;  
    private String nome;
```

```
    Aluno(int a, String b) {  
        this.codigo = a;  
        this.nome = b;  
    }
```

```
    protected void escrevaAluno() {  
        System.out.println("Codigo: " + this.codigo);  
        System.out.println("Nome: " + this.nome);  
    }
```

Observe que os atributos desta classe estão privados, forçando a utilização de métodos para definir seus valores.

Veja nesta classe a definição do construtor para inicializar os dois atributos.

Um método que exibe os atributos da classe também foi definido.

Acesso a Classe Pai (cont.)

```
final class AlunoOnline extends Aluno {  
  
    String dataacesso;  
  
    AlunoOnline(int a, String b, String c) {  
        super(a, b);  
        this.dataacesso = c;  
    }  
  
    public void relatorioAcesso() {  
        super.escrevaAluno();  
        System.out.println("O último acesso foi em " +  
this.dataacesso);  
    }  
}
```

Repare na referência a classe pai na criação da nova subclasse.

Observe que no construtor desta subclasse, a primeira instrução chama o construtor da classe pai, através da palavra **SUPER**. Desta forma, não há a necessidade de reescrever o código do construtor pertencente à classe pai.

Neste ponto, um método da classe pai foi chamado, também através da palavra **SUPER**.

Acesso a Classe Pai (cont.)

```
class Programa {  
  
    public static void main(String[] args) {  
  
        AlunoOnline x = new AlunoOnline(1, "Fulano",  
        "29/04/2010");  
  
        x.relatorioAcesso();  
  
    }  
}
```

Veja que neste ponto o construtor da subclasse recebe os valores dos atributos herdados e do atributo especializado.

Este método retornará os valores dos atributos informados através do construtor.

Exercício

Gerenciar Contas (parte 1)

- Elabore uma classe **abstrata** **ContaBancaria** com os seguinte membros:
 - atributo **nome**;
 - atributo **saldo**;
 - método **sacar**
 - método **depositar**

Exercício

Gerenciar Contas (parte 2)

- Acrescente ao projeto duas classe herdadas de ContaBancaria, com as seguintes características:
 - **ContaPoupanca:**
 - atributo **taxa_rendimento** (valor referente a quantos porcentos a poupança rende por mês)
 - método **calcularNovoSaldo** (atualiza o saldo com base na porcentagem de rendimento do mês)
 - **ContaEspecial:**
 - atributo **credito** (um valor de crédito extra);
 - métodos **solicitarCredito** (acrescenta o valor do crédito ao saldo);

Exercício

Gerenciar Contas (parte 3)

- Crie um programa principal contendo um método main, com as seguintes atribuições:
 - Pergunte ao usuário seu nome e seu saldo inicial;
 - Crie um novo objeto ContaPoupanca ou ContaEspecial passando o nome e o saldo do usuário através do método construtor;
 - Pergunte ao usuário quanto ele deseja sacar e utilize o método sacar();
 - Pergunte ao usuário quanto ele deseja depositar e utilize o método depositar();
 - Utilize o método calcularNovoSaldo ou solicitarCredito e depois exiba o novo saldo.

Veja a propriedade da orientação a objetos responsável pela modularização dos softwares

POLIMORFISMO

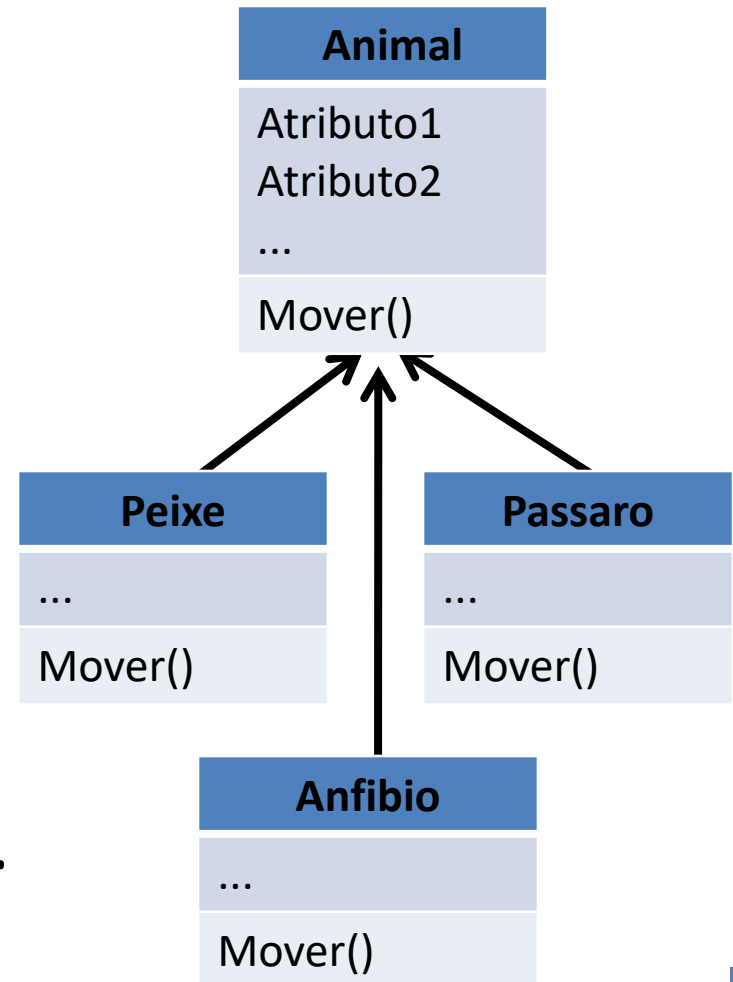
**Todo ser humano caminha da
mesma forma?**



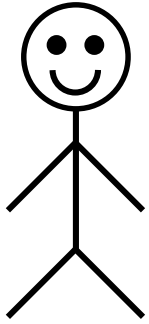
UNDB

Polimorfismo

- Propriedade da orientação a objeto que possibilita que subclasses invoquem métodos herdados através da mesma assinatura, embora comportem-se de maneira diferente.
 - A chamada de acesso ao método é a mesma, mas a implementação é diferente.



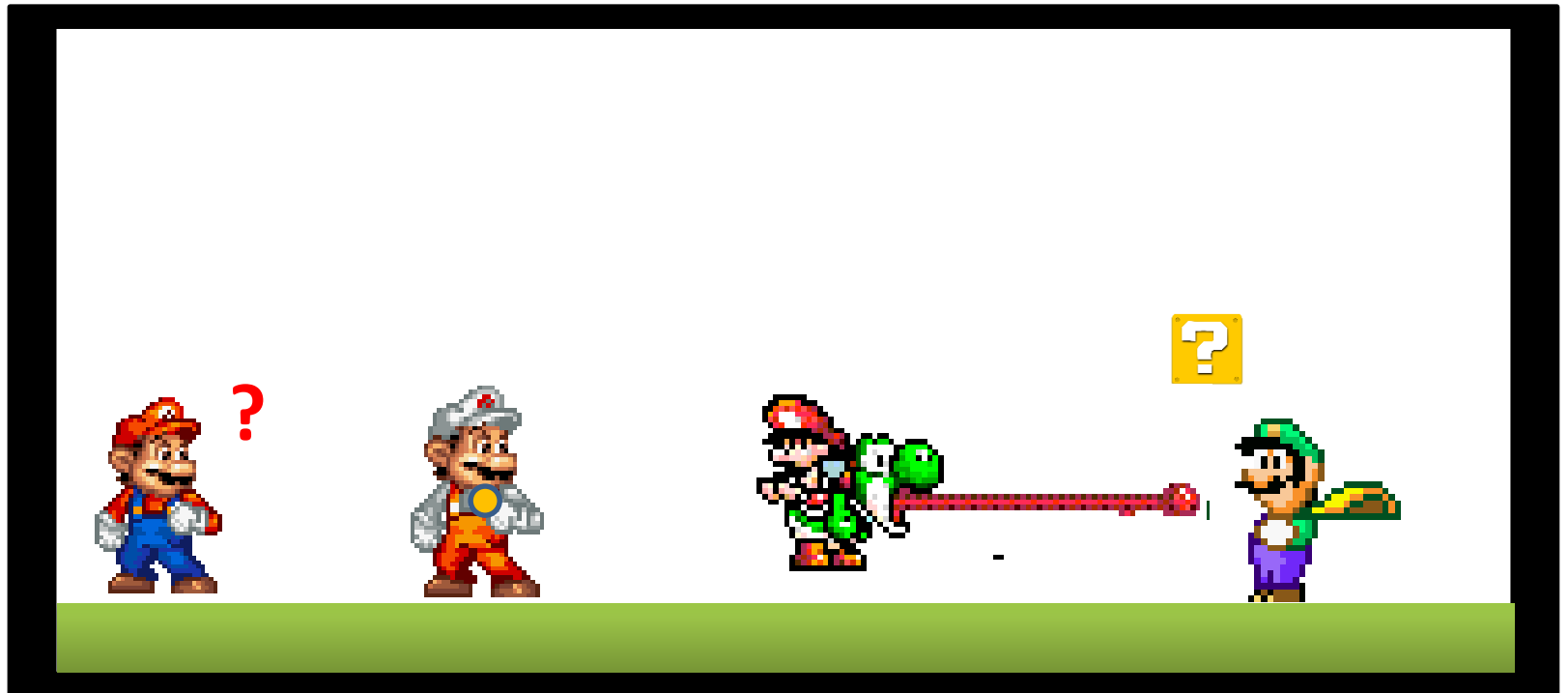
Polimorfismo (cont.)



Pressiona o botão de **AÇÃO**



Cada objeto executa a ação da forma apropriada àquele tipo de objeto.



Polimorfismo (cont.)

- Ao permitir a modificação de métodos herdados de um mesmo pai, o polimorfismo:
 - Torna os sistemas facilmente **extensíveis**, pois novas classes podem ser adicionadas com pouca ou nenhuma modificação a partes gerais do sistema.
 - **Padroniza** as operações realizadas por classes semelhantes.



Como utilizar Polimorfismo na programação?



Polimorfismo (cont.)

- O polimorfismo é obtido através da definição de várias implementações para um mesmo método, o qual pode estar em uma única classe ou em classes distintas.
- Existem duas formas de implementação do polimorfismo:
 - **Sobrecarga** (*Overload*): Nessa forma, vários métodos são declarados, todos com o mesmo nome, mas cada um desses métodos possuirão assinaturas distintas.
 - **Sobrescrita** (*Override*): Possível apenas em classes herdadas. Nessa forma, o método herdado é sobrescrito (substituído) na subclasse, mantendo a mesma assinatura em ambas as classes.

Polimorfismo (cont.)

```
class Funcionario {  
    protected int codigo;  
    protected String nome;  
    protected double salario;  
  
    Funcionario(int a, String b, double c) {  
        this.codigo = a;  
        this.nome = b;  
        this.salario = c;  
    }  
    protected double getSalario(){  
        return this.salario;  
    }  
    protected double getSalario(double bonus){  
        return this.salario + bonus;  
    }  
}
```

Veja nesta classe a definição do método de acesso ao salário, tanto sem parâmetros de entrada quanto recebendo valores externos. Por tratar-se de dois métodos de assinaturas distintas em uma mesma classe, é um exemplo de **SOBRECARGA**.

Polimorfismo (cont.)

```
final class Vendedor extends Funcionario {
```

```
    public double comissao;
```

```
    Vendedor(int a, String b, double
```

```
        super(a, b, c);
```

```
        this.comissao = d;
```

```
}
```

```
    protected double getSalario() {
```

```
        return (super.salario + this.comissao);
```

```
}
```

```
}
```

Veja a reescrita do método de recuperação do salário, que agora retorna a soma dos valores de dois atributos. Como a assinatura é a mesma da classe pai, esse método é um exemplo de **SOBRESCRITA**.

Polimorfismo (cont.)

```
class Programa {  
  
    public static void main(String[]  
  
        Funcionario a = new Funcionario(1, "Fulano",  
1500.00);  
        Vendedor b = new Vendedor(2, "Sicrano",  
2000.00, 700.00);  
  
        System.out.println(a.nome + " - Salário: " +  
a.getSalario());  
        System.out.println(b.nome  
b.getSalario());  
    }  
}
```

Repare que dois objetos foram definidos como sendo da classe pai, mas cada um deles referencia um objeto de uma classe/subclasse pertencente a essa hierarquia.

Observe que os dois objetos chama um método com o **MESMO NOME**. Entretanto, graças ao Polimorfismo, a implementação (execução) de cada método será diferente.

Todo método herdado pode ser sobrescrito?



Modificadores de Métodos

- É possível atribuir modificadores que afetam o comportamento dos métodos polimórficos:
 - **ABSTRACT**: Indica que a implementação de um método não será definido na classe a qual esse método pertence, mas sim nas subclasses, tornando sua implementação OBRIGATÓRIA.
 - **FINAL**: Indica que o método não poderá ser reescrito, tornando o método NÃO polimórfico.

Exemplo de Métodos Abstratos e Final

Quando houve métodos abstratos, a classe também deve ser abstrata, informando que a mesma não será instanciada.

```
abstract class FigGeometrica {
```

```
    protected double area;
```

```
    protected double perimetro;
```

```
    abstract protected double calcularArea();
```

```
    abstract protected double calcularPerimetro();
```

```
}
```

Observe que a classe possui dois métodos sem nenhum código, graças ao modificador **ABSTRACT**. A implementação desses métodos fica a cargo das subclasses.

Exemplo de Métodos Abstratos e Final (cont)

```
class Retangulo extends FigGeometrica {  
    public double altura;  
    public double base;
```

Observe a reescrita dos métodos definidos na classe pai.

```
    Retangulo(double a, double b){  
        this.altura = a;  
        this.base = b;
```

```
    }  
    public double calcularArea(){  
        return this.base * this.altura;  
    }
```

Veja que este método está com o operador **FINAL**, informando que o mesmo não será modificado nas subclasses.

```
    final public double calcularPerimetro(){  
        return (2 * this.base) + (2 * this.altura);  
    }
```


Exemplo de Métodos Abstratos e Final (cont)

```
final class Quadrado extends Retangulo {  
  
    Quadrado(double a) {  
        super(a, a);  
    }  
  
    public double calcularArea() {  
        return Math.pow(this.base,  
    }  
  
    public double calcularPerimetro() {  
        return 4 * this.base;  
    }  
}
```

Observe a reescrita do método definido na classe pai.

Este método **NÃO** pode ser implementado e gerará um erro, uma vez que na classe pai o método está com o operador **FINAL**.

Não Confundir!!

Classe

- ***Abstract*** – Indica que não pode ser instanciada;
- ***Final*** – Indica que não pode ser herdada;
- Relacionada aos conceito de herança.

Método

- ***Abstract*** – Indica que deve ser sobrescrito;
- ***Final*** – Indica que não pode ser sobrescrito;
- Relacionado aos conceitos de polimorfismo.

Exercício 1

- A classe abstrata **Casa** possui um endereço, um preço e métodos para imprimir cada um desses atributos. Assim, crie a classe **CasaNova**, que é uma subclasse de **Casa**, e sobrescreva o método de imprimir preço fazendo-o exibir um adicional de 10% ao valor real. Além disso, crie uma classe **CasaAntiga**, que também é subclasse de **Casa**, mas que sobrescreve o método de imprimir o endereço substituindo-o pela exibição da mensagem “Endereço Desconhecido!”.

Exercício 2

- A classe **Ingressos** possui atributos referente a um evento (nome e local), o valor do ingresso e um método para imprimir todas essas informações;
- A classe **Vip** é um **Ingresso** e possui informações adicionais como se a festa é openBar e o número do camarote;
- Na classe **Vip**, sobrescreva o método herdado de imprimir ingresso para que ele imprima os valores de openBar e numero do camarote também.
- O que mudaria se o método de imprimir da classe **Ingresso** fosse abstrato? Como ficaria o código final?

Exercício 3

- Crie uma classe **Restaurante** que possui métodos:
 - **realizarPedido()** – exibe na tela que o usuário ainda não fez o pedido.
 - **realizarPedido(String comida)** – exibe na tela o nome da comida que o usuário passou por parâmetro;
 - **realizarPedido(String comida, String bebida)** – exibe na tela o nome da comida e bebida solicitados;
 - **realizarPedido(String comida, int quantidade)** – exibe na tela o nome da comida e a quantidade solicitada;
 - **realizarPedido(int quantidade, String bebida)** – exibe na tela o nome da bebida e a quantidade solicitada;
- Crie um objeto da classe Restaurante e teste as várias formas de chamar o mesmo método.

Exercício

- A classe SuperMario possui o método **usarHabilidade()** e é capaz de executar uma habilidade diferente dependendo da qual subclasse acionou o método. Como podemos implementar esse polimorfismo do método usarHabilidade()?

