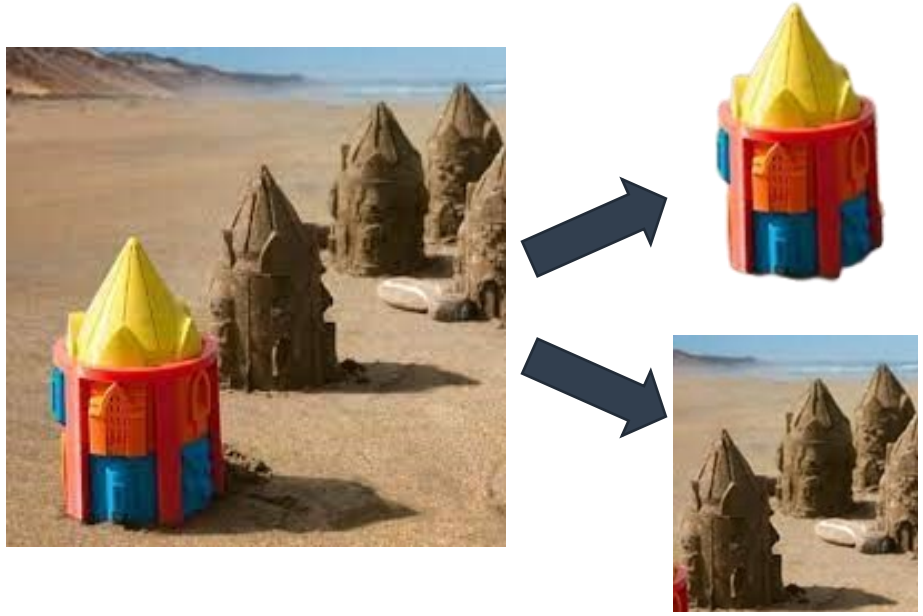
AED - Algoritmos e Estruturas de Dados

Aula 3

Prof. Rodrigo Mafort

Conceito de Classe

- Nos exemplos anteriores, todos os cachorros tem os mesmos atributos e métodos...
- Como definir esse modelo para todos os cachorros?
- Classes são como moldes ou formas para construir objetos



Classe Castelo de Areia

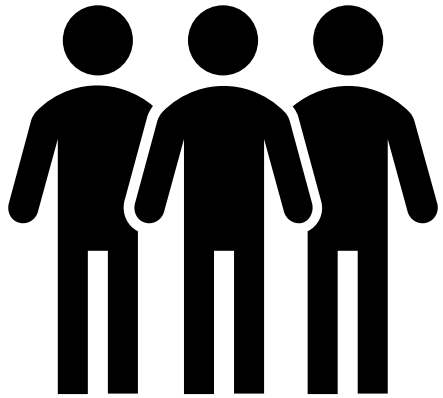
Objetos Castelo de Areia

Aula de Hoje: Como definir uma classe

- Vamos criar uma classe para representar pessoas:
  - Cada pessoa tem Nome, CPF e Idade
  - Cada pessoa pode se apresentar (Olá, sou ....) e falar sua idade (Tenho .... anos)

Pessoa
Nome CPF Data de Nascimento
Se Apresentar Falar Idade

- Depois, vamos criar 3 pessoas.
- Peça que cada uma diga seu nome e sua idade

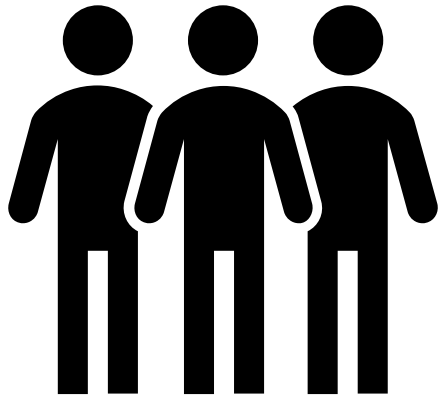


Aula de Hoje: Como definir uma classe

- Classe Pessoa:

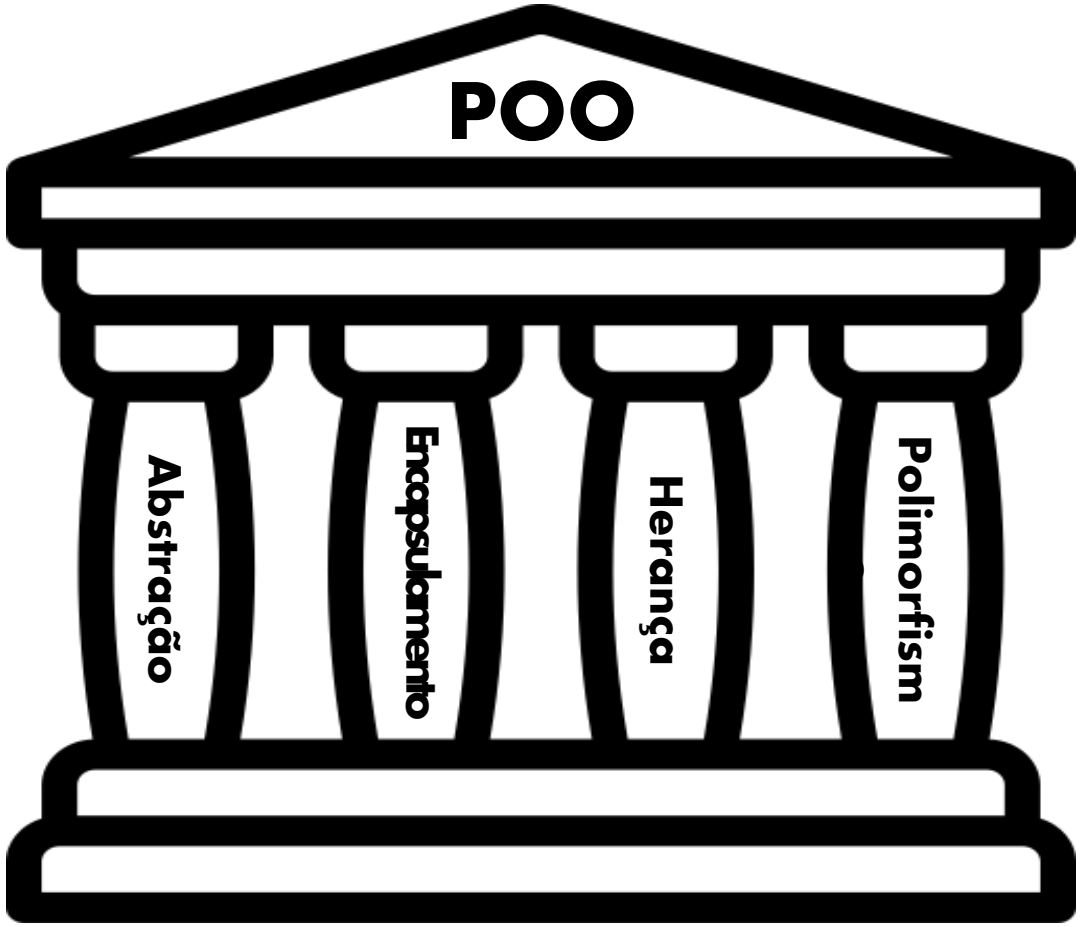
- Quais são os atributos que cada pessoa deve ter?
  - Nome
  - CPF
  - Data de Nascimento

Pessoa
Nome CPF Data de Nascimento
Se Apresentar Falar Idade



Aula de Hoje: Como definir uma classe

- Antes de implementar os atributos precisamos entender dois conceitos:
  - Abstração
  - Encapsulamento.



Abstração

- Significado (Huaiss): Operação intelectual em que um objeto de reflexão é isolado de fatores que comumente lhe estão relacionados na realidade.

- Mas o que significa isso na programação?

- Vamos analisar as entidades que estamos modelando:
  - O que esse objeto irá realizar no nosso sistema?
  - Quais são os atributos que precisamos implementar?
  - Quais são as ações que esse objeto irá executar no sistema?

Abstração

- Reduzimos o problema ao necessário para o contexto.
- Removemos detalhes que não são significativos para o nosso sistema.
- Ao remover esses detalhes, focamos no que é essencial e reduzimos a complexidade de lidar com o problema.
- Ignoramos detalhes que não são significamos.
- Assim podemos nos concentrar no que é relevante para a compreensão e solução do problema.

Abstração

- Como vamos usar o conceito de abstração na programação orientada a objetos:
  - Vamos identificar e isolar as classes relevantes para o problema em questão
  - Ao modelar essas classes, vamos focar nas características que são significativas para o problema. Não precisamos considerar ou modelar todas as complexidades e detalhes dessas entidades.
  - Ao focar no que importa:
    - Reduzimos a complexidade do sistema.
    - Facilitamos a implementação

Encapsulamento

- A ideia do encapsulamento é proteger (muitas vezes de nós mesmos) atributos e métodos dos nossos objetos.

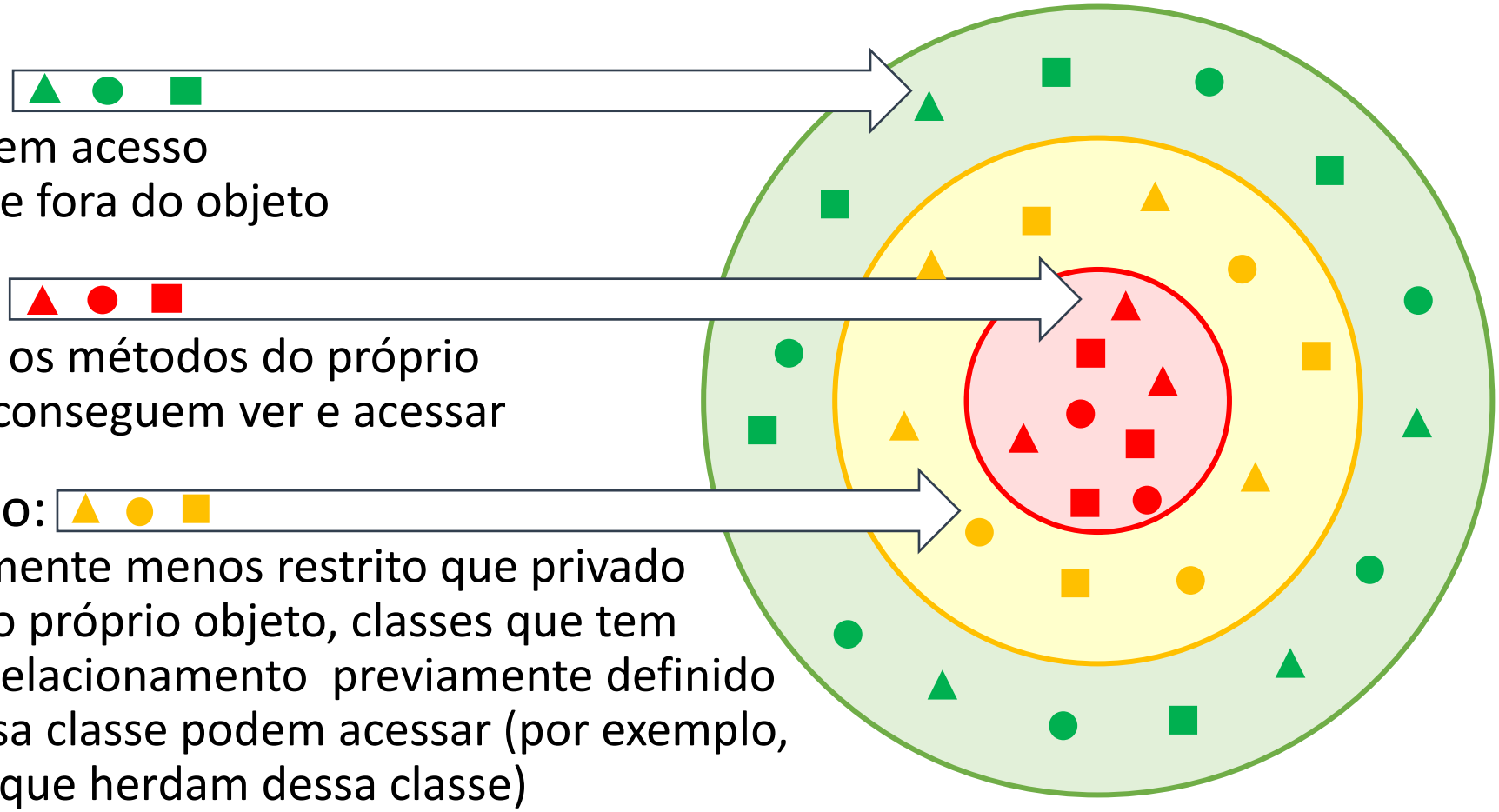
- Por exemplo:
  - Suponha uma classe que modele uma conta bancária.
  - A conta tem um atributo saldo.
  - Faz sentido permitir que esse saldo seja alterado diretamente?
  - Ou ele deveria ser alterado por meio de depósitos e saques?

- Com o encapsulamento, podemos esconder essa informação de forma que ela só possa ser manipulada de dentro do objeto, por meio de métodos específicos.

Níveis de Acesso

- A ideia do encapsulamento é definir três níveis de proteção (acesso) aos atributos e métodos:

- Público:
  Todos tem acesso
  Dentro e fora do objeto
- Privado:
  Apenas os métodos do próprio objeto conseguem ver e acessar
- Protegido:
  Ligeiramente menos restrito que privado
  Além do próprio objeto, classes que tem algum relacionamento previamente definido com essa classe podem acessar (por exemplo, classes que herdam dessa classe)

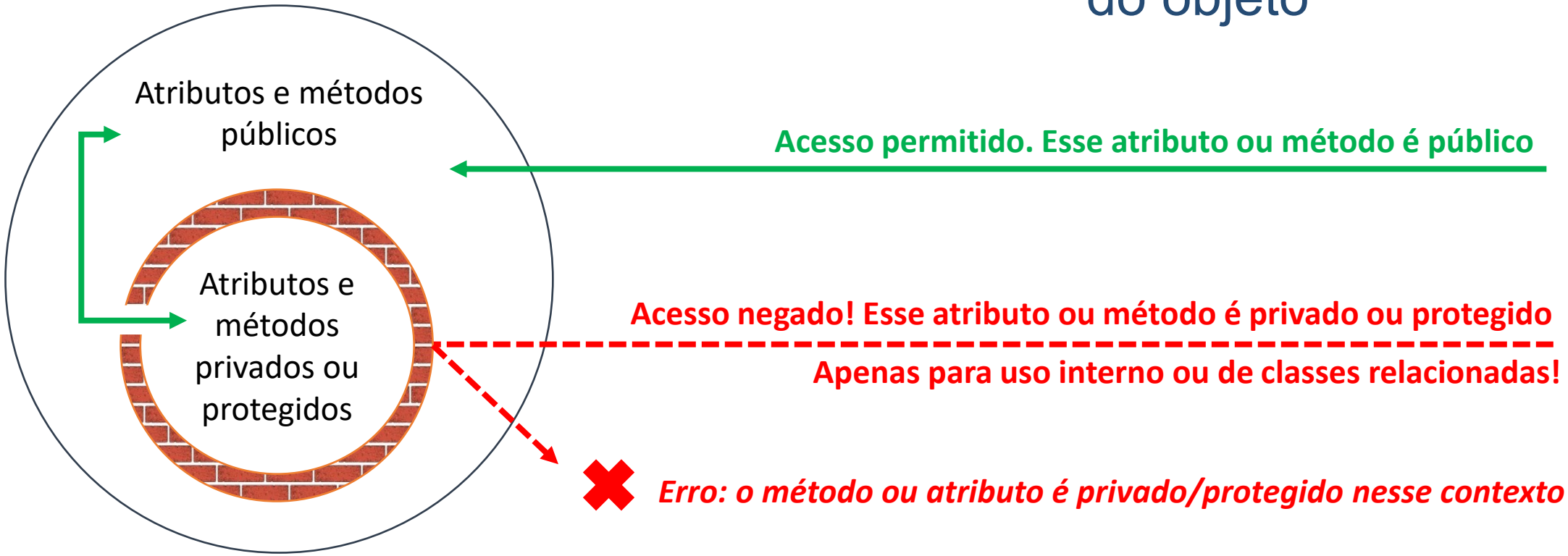


Encapsulamento

- Proteger partes do objeto de acesso externo

Ao definir a classe

Ao acessar o método/atributo do objeto



Encapsulamento

- Usando o encapsulamento podemos:
  - Esconder/omitir informações importantes apenas para o funcionamento interno de um objeto
  - Deixar acessíveis apenas os atributos e métodos necessários para interação com o mundo externo ao objeto

- Para lembrar: A meta do encapsulamento é reduzir pontos de falha.
- Proteger métodos e atributos de um eventual acesso indevido.

Exemplo:

- Considere que você está implementando uma classe para lidar com a autenticação de um sistema.
- É uma boa política permitir acesso direto ao atributo senha?
- Não seria melhor implementar um método que criptografasse a senha?
- Desta forma, a senha só poderia ser acessada por métodos da classe usuário.

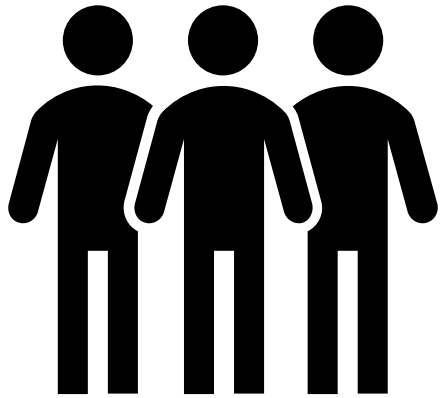
Usuário
Nome Login Senha
EfetuarLogin EfetuarLogout CriptografarSenha DescriptografarSenha

Classe Pessoa

- Quais são os atributos que cada pessoa deve ter?
  - Nome
  - CPF
  - Data de Nascimento

- Por enquanto, vamos considerar que todos os atributos são **Públicos**.

Pessoa
Nome CPF Data de Nascimento
Se Apresentar Falar Idade



Classe Pessoa (Pessoa.h)

```
#ifndef PESSOA_H
#define PESSOA_H

#include <string>
using namespace std;

class Pessoa
{
    public:
        string Nome;
        string CPF;
        int Idade;
        void SeApresentar();
        void FalarIdade();
    protected:
    private:
};

#endif // PESSOA_H
```

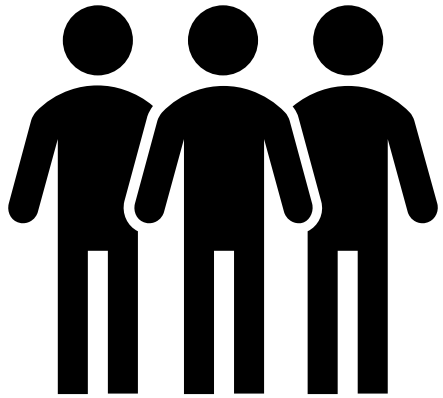
Classe Pessoa

- Quais são os métodos que cada pessoa deve ter?
  - Se Apresentar
  - Falar Idade

- Assim como os atributos, podemos definir os métodos como Públicos, Protegidos e Privados.

- Por enquanto, vamos considerar que todos os métodos são **Públicos**.

Pessoa
Nome CPF Data de Nascimento
Se Apresentar Falar Idade



Classe Pessoa (Pessoa.h)

```
#ifndef PESSOA_H
#define PESSOA_H

class Pessoa
{
    public:
        string Nome;
        string CPF;
        int idade;
        void SeApresentar();
        void FalarIdade();
    protected:
    private:
};

#endif // PESSOA_H
```

Classe Pessoa (Pessoa.cpp)

- Sempre que definimos os métodos no arquivo .h, precisamos apresentar a implementação desse método no arquivo .cpp

- Arquivo .h: Somente o protótipo da função

```
<tipo do retorno> <nome do método> (<parâmetros >);
```

- Arquivo .cpp: Implementação dos métodos:

```
<tipo do retorno> <nome da classe>::<nome do método>
                                        (<parâmetros>)
{
     ...
}
```

Operador ::

- Usado para definir o escopo

- “O que pertence ao que” ou “o que está contido onde”

- Por exemplo:
  - `<tipo do retorno> <classe>::<método>(<parâmetros>)`
  - O método **<método>** pertence a classe **<classe>**

  - **`std::cout`**
  - A função **`cout`** está definida no namespace **`std`**

Classe Pessoa (Pessoa.cpp)

```
#include "Pessoa.h"
#include <iostream>

void Pessoa::SeApresentar()
{
    cout << "Ola, sou " << Nome << "." << endl;
}

void Pessoa::FalarIdade()
{
    cout << "Tenho " << Idade << " anos." << endl;
}
```

Métodos

```
void Pessoa::SeApresentar()
{
    cout << "Ola, sou " << Nome << "." << endl;
}
```

- Onde **Nome** foi definido?
- Ao que **Nome** se refere?

- Ao objeto cujo método SeApresentar() foi executado!
- O método não é de um objeto? Esse objeto não tem atributos?
- O método acessa os atributos do objeto! Eles são uma entidade única: um objeto. (Observe que não passamos nenhum parâmetro!)

Como instanciar objetos

- Agora que definimos nossa primeira classe, vamos fazer alguns testes.

- Primeiro, vamos instanciar um objeto dessa classe:

```
Pessoa Joao;
```

- Depois vamos atribuir valores aos atributos do objeto:

```
Joao.Nome = "Joao das Couves";
Joao.Idade = 20;
```

- Em seguida, vamos executar um método desse objeto:

```
Joao.SeApresentar();
```

main.cpp

```
#include <iostream>

#include "Pessoa.h" //importe o arquivo Pessoa.h (onde a classe foi definida)

using namespace std;

int main()
{
    Pessoa Joao;      //Instancie um objeto da classe Pessoa
    Joao.Nome = "Joao das Couves"; //Atribua um valor ao atributo Nome do objeto Joao
    Joao.Idade = 20; //Atribua um valor ao atributo Idade do objeto Joao
    Joao.SeApresentar(); //Chame/Execute o método SeApresentar do objeto Joao

    return 0;
}
```

Esse código ainda não funciona...

Construtores e Destrutores

- Podemos criar métodos especiais que são chamados sempre que um objeto da classe é instanciado (criado) ou destruído (liberado da memória).

- Os métodos construtores permitem inicializar os atributos com informações (tal como inicializamos contadores e acumuladoras)
- Podemos definir vários construtores, com diferentes combinações de parâmetros.
- Uma classe sem construtores não pode ser instanciada (construída).

Construtores e Destrutores

- Já os destrutores são utilizados para liberar a memória de objetos e estruturas alocadas dinamicamente (comando malloc).
- Sempre que um objeto vai ser destruído, o método destrutor é executado.
- Se não houver nenhum objeto (ou estrutura alocada) que precise ser liberado manualmente, o destrutor não precisa ser implementado.

- Métodos construtores e destrutores não tem nenhum retorno (nem void).

Pessoa.h (somente a definição da classe)

```
class Pessoa
{
    public:
        string Nome;
        string CPF;
        int Idade;
        void SeApresentar();
        void FalarIdade();

        Pessoa();
        Pessoa(string Nome, int Idade);

        ~Pessoa();
};
```

Construtores

Destrutor

Construtores de Pessoa

- O primeiro construtor definido não especifica nenhum parâmetro.
- Esse tipo de construtor vazio permite criar objetos e depois atribuir informações (como fizemos no exemplo da Main).
- Caso não seja necessário inicializar nenhum atributo, esse construtor pode ficar em branco (sem nada dentro do método).

- Já o segundo construtor define que podemos também instanciar um objeto da classe Pessoa já atribuindo valores aos atributos Nome e Idade.

Pessoa.cpp (os construtores)

```
Pessoa::Pessoa()
{
    //Não faz nada...
}

Pessoa::Pessoa(string nome, int idade)
{
    Nome = nome;
    Idade = idade;
}
```

main.cpp

```
#include <iostream>

#include "Pessoa.h" //importe o arquivo Pessoa.h (onde a classe foi definida)

using namespace std;

int main()
{
    Pessoa Joao;      //Instancie um objeto da classe Pessoa
    Joao.Nome = "Joao das Couves"; //Atribua um valor ao atributo Nome do objeto Joao
    Joao.Idade = 20; //Atribua um valor ao atributo Idade do objeto Joao
    Joao.SeApresentar(); //Chame/Execute o método SeApresentar do objeto Joao

    return 0;
}
```

Qual construtor é executado?

main.cpp

```
#include <iostream>

#include "Pessoa.h"

using namespace std;

int main()
{

    Pessoa Joao("Joao das Couves",20);
    Joao.SeApresentar();

    Pessoa Pedro("Pedro das Pedras",50);
    Pedro.SeApresentar();

    return 0;
}
```

Qual construtor é executado?

Resumo:

- Até agora já sabemos:
  - Conceitos de POO
  - Como definir classes em C++
  - Como criar atributos e métodos
  - Como instanciar objetos
  - Conceitos de construtores e destrutores

Exemplo

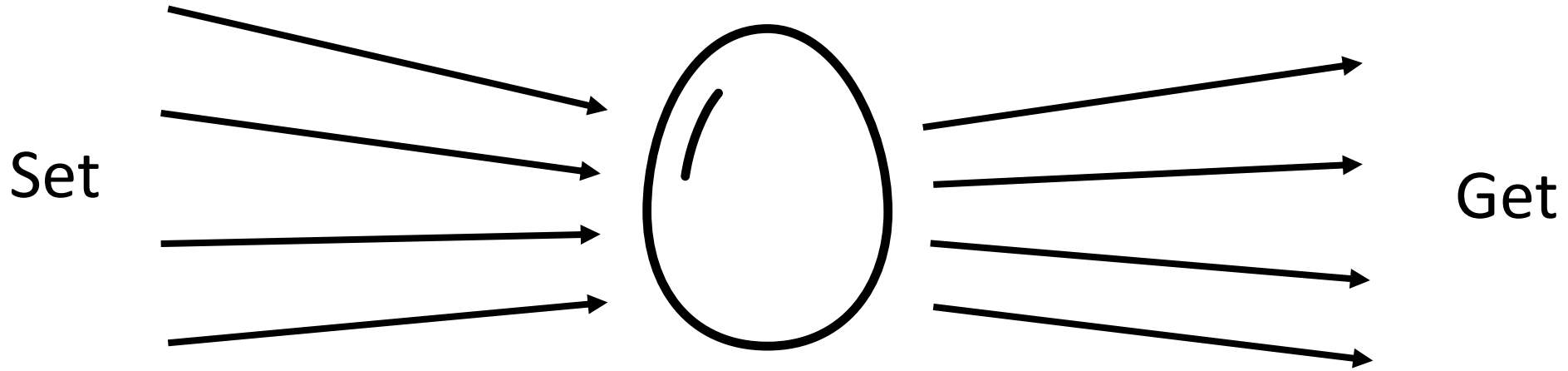
- Crie uma classe que implemente um círculo.
- Atributos: raio
- Métodos: obterRaio, calcularArea e calcularPerimetro

- Construtor: o construtor deve receber e inicializar o raio
- O raio deve ser somente leitura. Uma vez definido, o raio não pode ser alterado. O raio pode ser apenas consultado.

Encapsulamento: Getter e Setter

- Muitas vezes queremos apenas proibir a escrita na variável, sem impedir a leitura...
- Ou queremos apenas validar a informação armazenada (se uma data é válida, se o CPF é válido, etc...)

- Para esses casos podemos usar getters e setters



Encapsulamento: Getter e Setter

- O lado get (obter) estabelece como vamos fazer a leitura do atributo
- O lado set (definir) estabelece como vamos fazer a escrita do atributo

Exemplos:

- Se não queremos permitir a escrita: Definimos apenas o get
- Se queremos validar os dados antes de salvar o atributo: Definimos um validador no lado set

- São formas de interagir com informações que queremos proteger no objeto.

Encapsulamento: Getter e Setter

```
class Pessoa
{
    private:
        string nome;
        int idade;
        string cpf;
        void SetCPF(string _cpf);

    public:
        Pessoa(string _cpf);
        Pessoa(string _nome,
         int _idade, string _cpf);
```

```
        void SetIdade(int _idade);
        void SetNome(string
                              _nome);

        string GetCPF();
        int GetIdade();
        string GetNome();
}
```

Encapsulamento: Getter e Setter

- Observe que no exemplo anterior o Setter de CPF é PRIVADO, logo só pode ser acessado de dentro da própria classe. Essa implementação possibilita que o Setter só seja executado pelo construtor, fazendo a validação do informação.

```
Pessoa::Pessoa(string _cpf)
{
    SetCPF(_cpf);
}
Pessoa::Pessoa(string _nome, int _idade, string _cpf)
{
    SetCPF(_cpf);
    SetNome(_nome);
    SetIdade(_idade);
}
```

Encapsulamento: Getter e Setter

- A vantagem dos Setters é permitir a validação da informação que será salva:

```
void Pessoa::SetCPF(string _cpf)
{
    if (_cpf.length() != 11)
        throw "CPF Inválido";

    for (int i = 0; i < 11; i++)
        if (!isdigit(_cpf[i))
            throw "CPF Inválido";

    cpf = _cpf;
}
```

Exercícios

1. Crie uma classe em C++ que represente uma coordenada no plano (X,Y). Implemente:
   a) A classe
   b) O construtor
   c) Um método que calcula a distância da coordenada atual até uma outra coordenada qualquer.

2. Faça exemplos de uso da classe e de cada método implementado

Exercícios

3. Implemente uma classe para listas sequenciais não ordenadas, tal como visto nas aulas.
   - Pense em qual nível de proteção (privado, protegido ou público) você deve criar cada estrutura necessária para manipular listas.
   - Implemente todos os métodos vistos em sala.

Exercícios

4. Pense, rascunhe e implemente classes para Cliente e Conta Bancária. Defina os atributos do cliente e da conta bancária. Suponha que é possível calcular saldo, depositar dinheiro e sacar dinheiro de uma conta bancária. Observe que não é possível depositar um valor negativo, nem sacar valores de uma conta sem dinheiro (zerada ou negativa).

5. Pense em como podemos relacionar as classes Cliente e Conta Bancária?
   - Cliente tem conta bancária?
   - Conta bancária tem cliente?

Relações entre Objetos

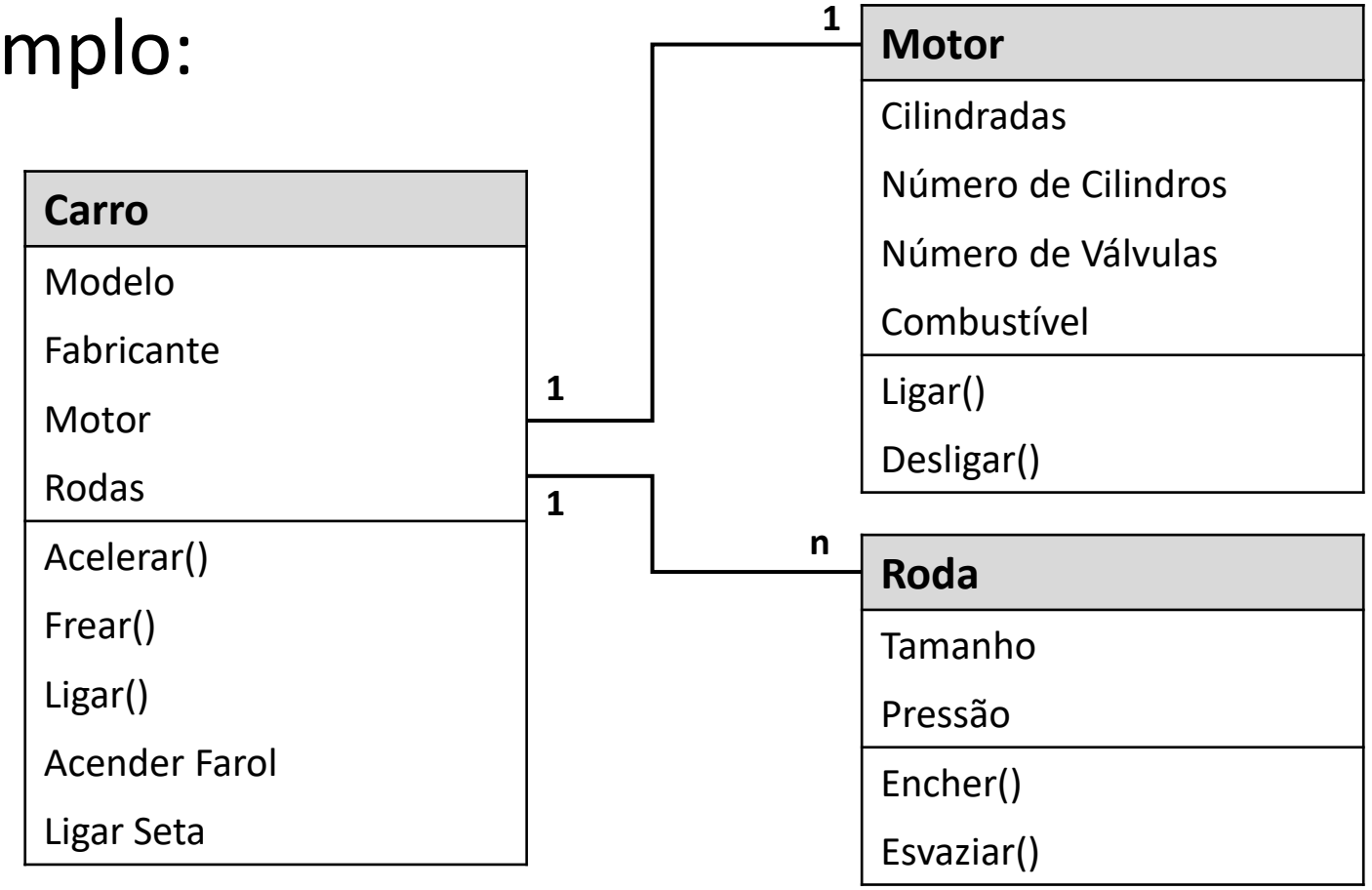
- Existem dois tipos de relações entre objetos: agregação e composição

- Agregação: Uma classe contém objetos de uma outra

- Composição: Difere da primeira pelo fato de que uma das classes não existe (ou faz sentido) sem a outra.

Agregação

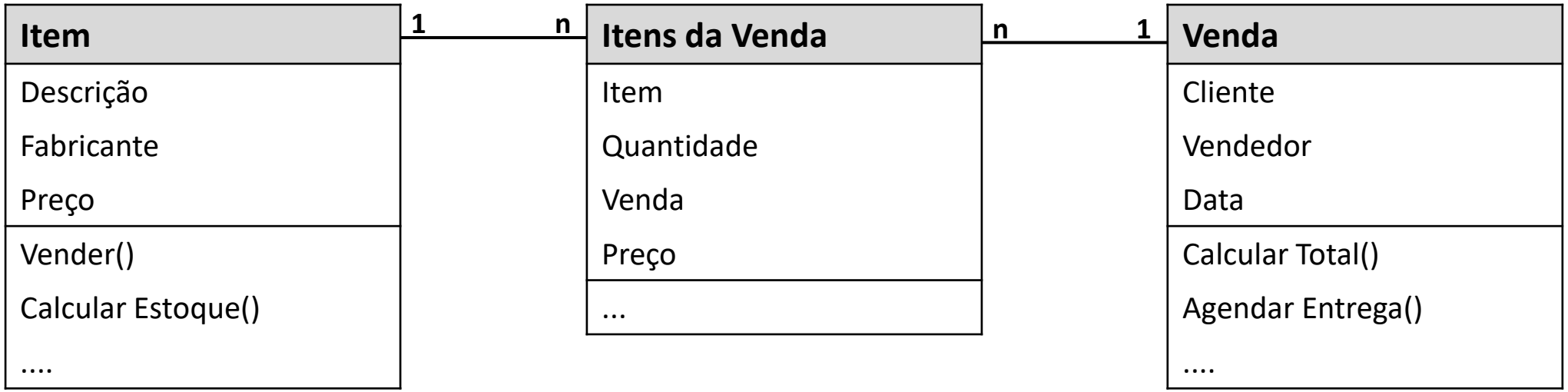
- Um objeto contém outros objetos (como atributos)
- Um objeto incorpora um outro (agrega)
- Exemplo:



- Podemos ter um motor e usar esse motor em um carro.
- Em seguida, usamos o mesmo motor em outro carro

Composição

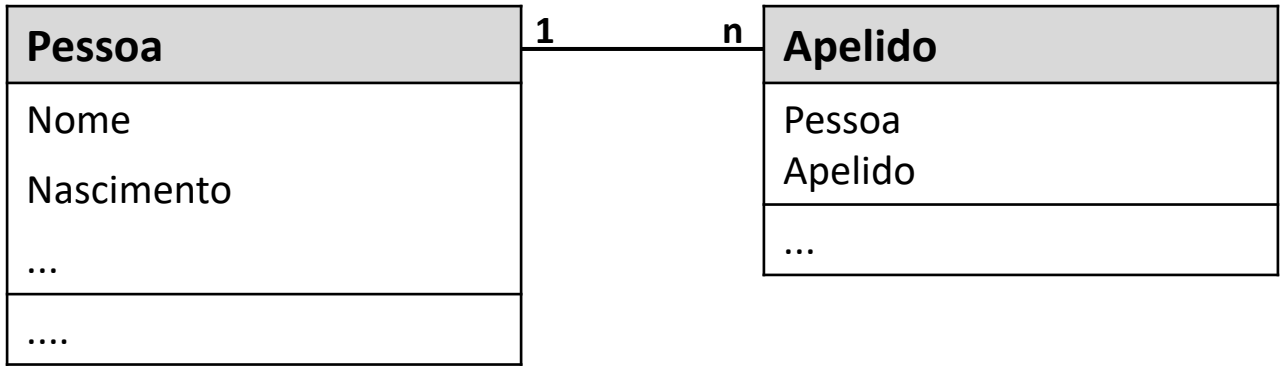
- Uma das classes depende da outra para existir
- Exemplo:



Itens da Venda só existe se houver um item e um pedido

Composição

- Uma das classes depende da outra para existir
- Outro exemplo:



Uma pessoa pode ter vários apelidos,

Mas um apelido não faz sentido sem uma pessoa

Repositório GitHub

- Para facilitar o envio das respostas das questões das aulas, criei um repositório no GitHub com os projetos

- https://github.com/rodrigomafort/ExemplosED-2024.1

- Cada questão foi implementada como um projeto do CodeBlocks.

Exercícios

1. Pense e implemente um modelo para um jogo de dados. O jogo é baseado em um número escolhido de dados customizados (com pelo menos 4 faces). Cada dado deve conter uma função que sorteia uma face. Além disso, seu jogo deve comportar vários jogadores, identificados por seus nomes. Cada jogador efetua uma aposta única: o número que ele acredita que será sorteado pelos dados (que não pode ser repetido pelos demais jogadores) e o valor da aposta. Depois que todos os jogadores fazem suas apostas, o jogo deve sortear o valor dos dados e apresentar o ganhador, que deverá receber a soma das apostas. Caso nenhum jogador acerte o número, o jogo deve avisar que a mesa ganhou o jogo (com o valor).

Exercícios

Modelo de Classes para o jogo

Dado
Número de Faces Face Sorteada
Sortear() Apresentar Face Sorteada

Jogador
Nome Número apostado Valor apostado
Definir Nome Fazer Aposta Mostrar Nome Apresentar Aposta

Jogo
Dados Jogadores
Lançar Dados Definir Ganhador Calcular Prêmio

Exercícios

2. Modele e implemente um cofrinho em C++:

   Considere que serão depositadas apenas moedas de 5, 10, 25 e 50 centavos e moedas de 1 real.

   Escreva métodos:

   - Para depositar as moedas
   - Para calcular o total depositado no cofrinho
   - Para sacar um valor do cofrinho
   - Para verificar se uma moeda é falsa (considere que moedas com valores diferentes dos descritos é considerada falsa)

   Atenção: Não é permitido alterar o número de moedas nem o valor do cofrinho sem usar os métodos descritos. Ao depositar, confira primeiro se a moeda não é falsa. Caso seja falsa, dispare uma exceção.

Pilares da Orientação a Objetos

- Agora que já aprendemos o básico da orientação a objetos
- Vamos avançar até os principais fundamentos da Programação Orientada a Objetos
- Existem quatro pilares básicos em que a orientação a objetos se apoia:
  - Abstração
  - Encapsulamento
  - Herança
  - Polimorfismo