

Programação e Desenvolvimento de Software 2

Programação Orientada a Objetos (Encapsulamento)

Prof. Luiz Chaimowicz
(slides adaptados do Prof. Douglas Macharet)

Introdução

- Abstração
 - Simplificação de um problema difícil
 - Representar características **essenciais** sem incluir os detalhes

O quê isso representa e faz?
- Ocultação de dados
 - Algumas informações devem ser escondidas do mundo externo
 - Desvincular a especificação da sua **implementação**
 - Usuário do TAD x Programador do TAD

Como ele faz?

Introdução

▪ Encapsulamento

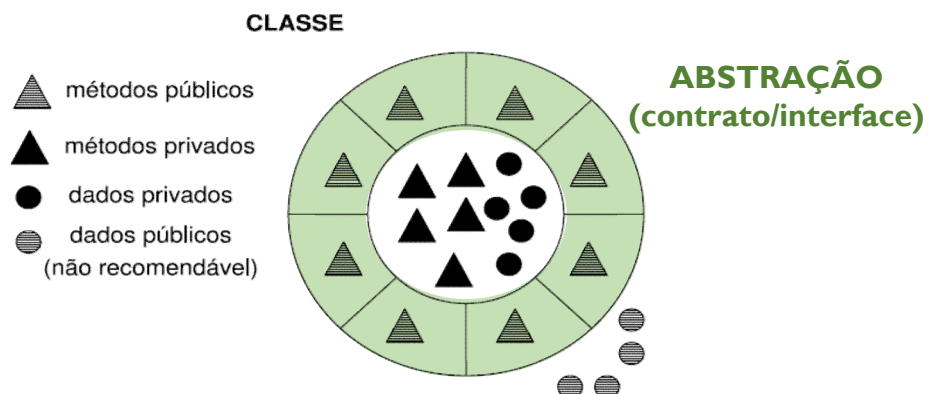
- Mecanismo que coloca juntos os dados e suas funções associadas, mantendo-os controlados em relação ao seu nível de acesso (visibilidade)



▪ Proporciona abstração

- Separa a visão externa da visão interna
- Utilizar sem conhecer/entender o código

Introdução



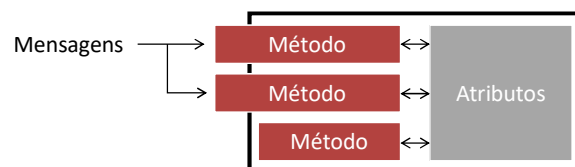
Encapsulamento

Benefícios

- **Desenvolvimento**
 - Melhor compreensão de cada classe
 - Facilita a realização de testes
 - Protege a integridade dos dados do Objeto
 - Reduz a chance de erros por parte do “usuário”
- **Manutenção/Evolução**
 - Interface deve ser o mais constante possível
 - Impacto reduzido ao modificar uma classe

Encapsulamento

- Encapsulamento ocorre nas classes
- Interface (contrato) e comportamento definidos pelos membros



- Definição baseada em modificadores de acesso

Encapsulamento

C++

- Modificadores de acesso
 - Public
 - Protected
 - Private
- Membros declarados após o modificador
- Checagem em tempo de compilação

<https://en.cppreference.com/w/cpp/language/access>

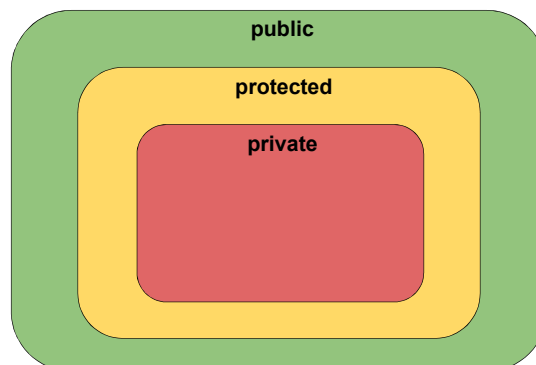


PDS 2 - Programação Orientada a Objetos (Encapsulamento)

7

Encapsulamento

Modificadores de acesso



PDS 2 - Programação Orientada a Objetos (Encapsulamento)

8


Encapsulamento

Modificadores de acesso – Public

- Os membros definidos neste escopo podem ser acessados de qualquer outra parte do código (interno/externo)
- Mais liberal dos modificadores
 - Fazem parte (definem) o contrato da classe
 - Deve ser usado com responsabilidade!
 - De preferência apenas métodos devem ser públicos

Encapsulamento

Modificadores de acesso – Public


Membros públicos.

```

class Ponto {
    public:
        int x;
        int y;

        Ponto(int x, int y) {
            this->x = x;
            this->y = y;
        }
};
  
```

Encapsulamento

Modificadores de acesso – Protected

- Os membros definidos neste escopo podem ser acessados dentro da própria classe e por outras classes que
 - Fazem parte da hierarquia (derivadas) ←
 - Com acesso público ou protegido
 - Classes “amigas” (*friends*)
 - Algo bem específico em C++
- Membros um pouco mais “reservados” que os públicos

Encapsulamento

Modificadores de acesso – Protected

```
class Base {
    protected:
        int i = 99;
};

class Derived : public Base
{
    public:
        int f() {
            i++;
            return i;
        }
};
```

Herança, será detalhado na próxima aula!

OK!

```
int main() {
    Base b;
    cout << b.i << endl; Erro!

    Derived d;
    cout << d.f() << endl;

    return 0;
}
```

Encapsulamento

Modificadores de acesso – Private

- Os membros definidos neste escopo podem ser acessados apenas por outros métodos da mesma classe (uso interno)
- O mais restritivo dos modificadores
 - Deve ser empregado sempre que possível
 - Utilizar métodos auxiliares de acesso
- Quando não há declaração explícita de um modificador
 - Classes → Padrão é privado
 - Structs → Padrão é público

Encapsulamento

Modificadores de acesso – Private

```
class Ponto {
    private:
        int _x;
        int _y;

    public:
        Ponto(int x, int y) : _x(x), _y(y) {}
};
```

Encapsulamento

Modificadores de acesso – Private

```
class Base {
    private:
        int i = 99;
};

class Derived : public Base
{
    public:
        int f() {
            i++; Erro!
            return i;
        }
};
```

```
int main() {
    Base b;
    cout << b.i << endl; Erro!

    Derived d;
    cout << d.f() << endl;

    return 0;
}
```

Encapsulamento

Modificadores de acesso – Private

Também podemos ter
Classes Privadas, dentro
de uma outra classe

```
class Ponto {
    private:
        class EstruturaPonto {
            public:
                double x;
                double y;
        };

        EstruturaPonto p;

    public:
        Ponto(int x, int y) {
            p.x = x;
            p.y = y;
        }
};
```


Encapsulamento

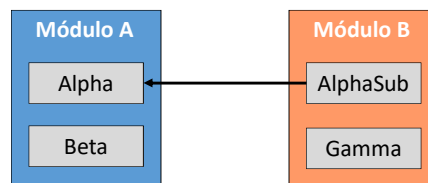
Modificadores de acesso

	Classe	Subclasse	Externo
Public	■	■	■
Protected	■	■	■
Private	■	■	■

Encapsulamento

Modificadores de acesso

- Visibilidade dos membros de **Alpha** de acordo com modificador de acesso



	Alpha	AlphaSub	Beta	Gamma
Public	■	■	■	■
Protected	■	■	■	■
Private	■	■	■	■

Encapsulamento

Acessando e modificando atributos

- Evitar a manipulação direta dos atributos
 - Acesso deve ser o mais restrito possível
 - De preferência todos devem ser private
- Sempre utilizar métodos auxiliares
 - Melhor controle das alterações
 - Acesso centralizado (programação defensiva)

Encapsulamento

Getters e Setters

- Convenção de nomenclatura dos métodos
- **Get**
 - Métodos usados apenas para a consulta (obter) dos atributos
- **Set**
 - Métodos usados para a alteração (definir) os atributos

Encapsulamento

Getters e Setters

```
class Ponto {
    private:
        double _x;
        double _y;

    public:
        Ponto(double x, double y) : _x(x), _y(y) {}

        void setX(double x) { this->_x = x; }
        void setY(double y) { this->_y = y; }

        double getX() { return this->_x; }
        double getY() { return this->_y; }
};
```

Encapsulamento

Getters e Setters

- Todos os atributos deveriam possuir get e set
- Nomenclatura alternativa
 - Atributos booleanos devem utilizar o prefixo “is”
 - Melhora a legibilidade e entendimento
- E uma coleção, possui ‘setColecao’?
 - Não!
 - Métodos auxiliares: adicionar, remover, ...

Encapsulamento

Getters e Setters

```
class Cliente {
    private:
        string _nome;
        bool _ativo;

    public:
        Cliente(string nome, bool ativo) : _nome(nome), _ativo(ativo) {}

        void setNome(string nome) { this->_nome = nome; }
        void setAtivo(bool ativo) { this->_ativo = ativo; }

        string getNome() { return this->_nome; }
        bool isAtivo() { return this->_ativo; }
};
```

Exercício

- Modelar uma conta bancária
 - Quais atributos devem existir?
 - Quais métodos devem existir?



Exercício

```
class Conta {

    public:
        int agencia;
        int numero;
        double saldo;

        Conta(int agencia, int numero) : agencia(agencia), numero(numero) {}

};
```

Exercício

```
class Conta {

    private:
        int _agencia;
        int _numero;
        double _saldo = 0.0;

    public:

        Conta(int agencia, int numero) : _agencia(agencia), _numero(numero) {}

        void setAgencia(int ag) { this->_agencia = ag; }
        void setNumero(int num) { this->_numero = num; }
void setSaldo(double saldo) { this->_saldo = saldo; }

        int getAgencia() { return this->_agencia; }
        int getNumero() { return this->_numero; }
        double getSaldo() { return this->_saldo; }

};
```

Exercício

```
class Conta {

    private:
        int _agencia;
        int _numero;
        double _saldo = 0.0;

    public:

        {...}

        void depositar(double valor) {
            this->_saldo += valor;
        }

        void sacar(double valor) {
            this->_saldo -= valor;
        }

};
```

Exercício

Descontando uma tarifa
em cada operação

```
class Conta {

    {...}

    public:

        {...}

        void depositar(double valor) {
            this->_saldo += valor;
            this->_saldo -= 0.25;
        }

        void sacar(double valor) {
            this->_saldo -= valor;
            this->_saldo -= 0.25;
        }

};
```

Exercício

```
class Conta {
    {...}

    public:

        {...}

        void depositar(double valor) {
            this->_saldo += valor;
            descontarTarifa();
        }

        void sacar(double valor) {
            this->_saldo -= valor;
            descontarTarifa();
        }

        void descontarTarifa() {
            this->_saldo -= 0.25;
        }
};
```

Exercício

```
class Conta {
    private:
        {...}

        void _descontarTarifa() {
            this->_saldo -= 0.25;
        }

    public:
        {...}

        void depositar(double valor) {
            this->_saldo += valor;
            descontarTarifa();
        }

        void sacar(double valor) {
            this->_saldo -= valor;
            descontarTarifa();
        }
};
```

Exercício

```
class Conta {

    private:
        {...}

        double const _TARIFA = 0.25;

        void _descontarTarifa() {
            this->_saldo -= _TARIFA ;
        }

    public:
        {...}

};
```

Exercício

```
class Conta {

    private:
        {...}


        static double constexpr _TARIFA = 0.25;

        void _descontarTarifa() {
            this->_saldo -= _TARIFA ;
        }

    public:
        {...}

};
```

Indica um atributo de classe com valor constante e conhecido durante a compilação.



Exercício

- Tarefas
 - Crie outras entidades relacionadas (Agencia, Cliente)
 - Pense em atributos e métodos para cada uma delas
 - Pratique o uso de diferentes modificadores de acesso
 - Faça a modularização de todas as classes (.hpp, .cpp)