>>> Programação Orientada a Objetos (POO)

Prof: André de Freitas Smaira

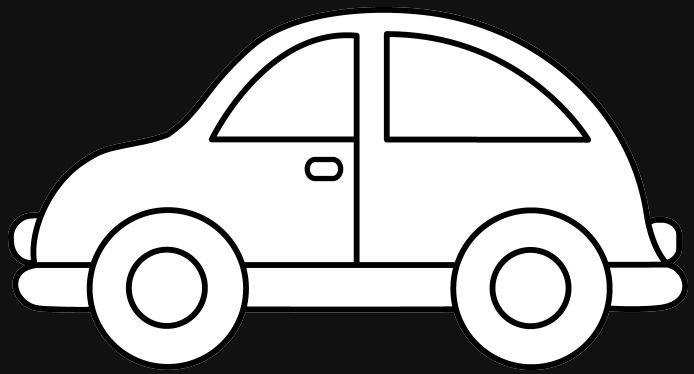
[~]$ _

>>> Introdução

[~]$ _

>>> Orientada a Objetos?

* O que é objeto? São objetos mesmo
* Com suas características e ações, veja:



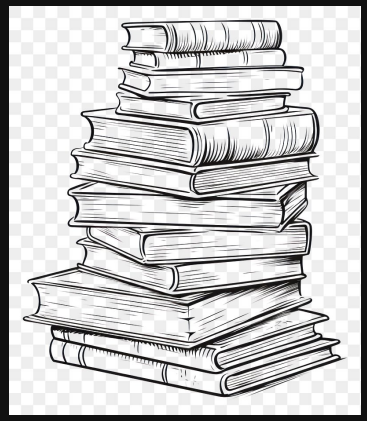
Características	Ações
Marca	Ligar
Cor	Acelerar
# Portas	Frear
# Passageiros	Buzinar

Características	Ações
Marca	Ligar
Processador	Recarregar
GPU	Mudar Brilho
RAM	Hibernar

[~]$ _

>>> Orientada a Objetos?

* Ou quem sabe algo mais relacionado à computação:



Características	Ações
Elementos	Inicializar
Tamanho	Inserir
Início	Remover
	Consultar frente

Características	Ações
Elementos	Inicializar
Tamanho	Inserir
	Remover
	Consultar topo

[~]$ _

>>> Orientada a Objetos?

* Necessidade de representarmos objetos
* Estamos falando apenas de abstração
* Por que POO?

[~]$ _

>>> Por que POO?


Programação
Estruturada
Dados
Função 1
Função 2
Função 3
...
Orientada a Objetos
Dados
Objeto 1
Método 1
Método 2
...
Dados
Objeto 2
Método 1
Método 2
...

[~]$ _

>>> Por que POO?

* POO ajuda no encapsulamento dos dados e da lógica em diferentes objetos
* Importante para grandes programas
* Facilita a manutenção
* Modularização => Fácil entendimento
* Basicamente vamos criar novos tipos de dados

[~]$ _

>>> Tipos e Operações

* Queremos essa relação indicada no código
* struct + funções não fornece essa indicação
* Tipo é caracterizado pelas **operações**, e **não pela implementação** (**interface**)
* O cliente não deve ter acesso à implementação
* Se cliente usa detalhes de implementação:
  * Fica difícil alterar implementação
  * Cliente pode alterar detalhes inconsistentemente
* O que é parte da **implementação**? E da **interface**?
  * **Implementação**: não interessam aos clientes
  * **Interface**: Determina o acesso pelo cliente
* **tipos <-> operações**
* Em POO por **classes**

[~]$ _

>>> Principais conceitos

* Classe
* Objeto
* Abstração
* Encapsulamento
* Herança
* Polimorfismo

[~]$ _

>>> Classe

* Descrição que **abstrai** o coisas da vida real, como carros e computadores
* Define o molde, ou template, para a criação de um **objeto**
* Separação o que é **interface** e o que é **implementação** é chamada **controle de acesso**

[~]$ _

>>> Controle de Acesso

* Implementação da classe:
  * Código da implementação
  * Membros privados (padrão)
* Membros públicos devem ser precedidos pelo rótulo public
* Membros públicos = interface

>>> Exemplo

```
class Rational {
    int _numerator, _denominator;
public:
    void set(int num, int den);
    void set(const Rational &r);
    int numerator();
    int denominator();
    Rational plus(const Rational &b);
    Rational minus(const Rational &b);
    Rational times(const Rational &b);
    Rational over(const Rational &b);
    double to_double();
};
```

[~]$ _

>>> Classes

* Dois tipos de membros:
    * Dados (ou campos)
    * Funções (ou métodos)
* Normalmente dados são privados
* Normalmente operações são públicas
* Classe => tipo de dados
* Variável => instância ou objeto da classe

[~]$ _

>>> Objeto

* Instância de uma classe
* Cada objeto possui características específicas
* Gol, Onix, Ka, HB20, QQ são possíveis exemplos de uma classe carro

[~]$ _

>>> Abstração

* Identificação dos **elementos relevantes** de um **objeto**, necessários para sua implementação

[~]$ _

>>> Acesso a Membros

```
class Rational {
    int _numerator, _denominator;
public:
    void set(int num, int den);
    void set(const Rational &r);
    int numerator();
    int denominator();
    Rational plus(const Rational &b);
    Rational minus(const Rational &b);
    Rational times(const Rational &b);
    Rational over(const Rational &b);
    double to_double();
};

int main() {
    Rational a, b, c;
    b.set(1,2);
    c.set(2,3);
    a.set(b.times(c));
    return 0;
}
```

>>> Implementação

```
void Rational::set(int num, int den) {
    _numerator = num;
    _denominator = den;
}

void Rational::set(const Rational &r) {
    _numerator = r._numerator;
    _denominator = r._denominator;
}

Rational Rational::times(const Rational &b) {
    Rational r;
    r.set(_numerator * b._numerator,
    _denominator * b._denominator);
    return r;
}
...
```

[~]$ _

>>> Compilação Separada

* Definição da classe (com protótipos dos métodos) no .h ou .hpp
* Definição dos métodos no .cpp
* Código cliente em arquivo separado

>>> Encapsulamento

* **Ocultação dos atributos** de uma classe para o programador (acessados apenas por métodos)
* **Implementação** fica **encapsulada** pela interface
* Facilita alterações de implementação, não afetando códigos cliente
* **Alterações na interface afetam clientes**, e devem ser evitadas

>>> CUIDADO COM PONTEIROS

* Membros ponteiros podem levar a falhas de segurança ou perda de encapsulamento
* Dados controlados pelo cliente devem ser copiados internamente
* Ao retornar ao cliente:
  * Fazer uma cópia, ou
  * Garantir que não pode ser alterado (const)

>>> Controle de Acesso

* Rótulos:
    * public: parte acessível ao cliente (interface)
    * private: parte não acessível ao cliente (implementação)

[~]$ _

>>> Diferença entre struct e class

public por padrão em **struct** e **private** por padrão em **class**

```
class A {
    int a;
public:
    void f(int x);
    int g();
};
```

```
class A {
public:
    void f(int x);
private:
    int a;
public:
    int g();
};
```

```
struct A {
public:
    void f(int x);
    int g();
private:
    int a;
};
```

```
class A {
private:
    int a;
public:
    void f(int x);
    int g();
};
```

```
class A {
public:
    void f(int x);
    int g();
private:
    int a;
};
```

```
struct A {
    void f(int x);
    int g();
private:
    int a;
};
```

[~]$ _