PEF – 5743 – Computação Gráfica Aplicada à Engenharia de Estruturas

Prof. Dr. Rodrigo Provasi

e-mail: provasi@usp.br

Sala 09 – LEM – Prédio de Engenharia Civil

Linguagens Orientadas à objetos

- Conceitos Básicos
- Características

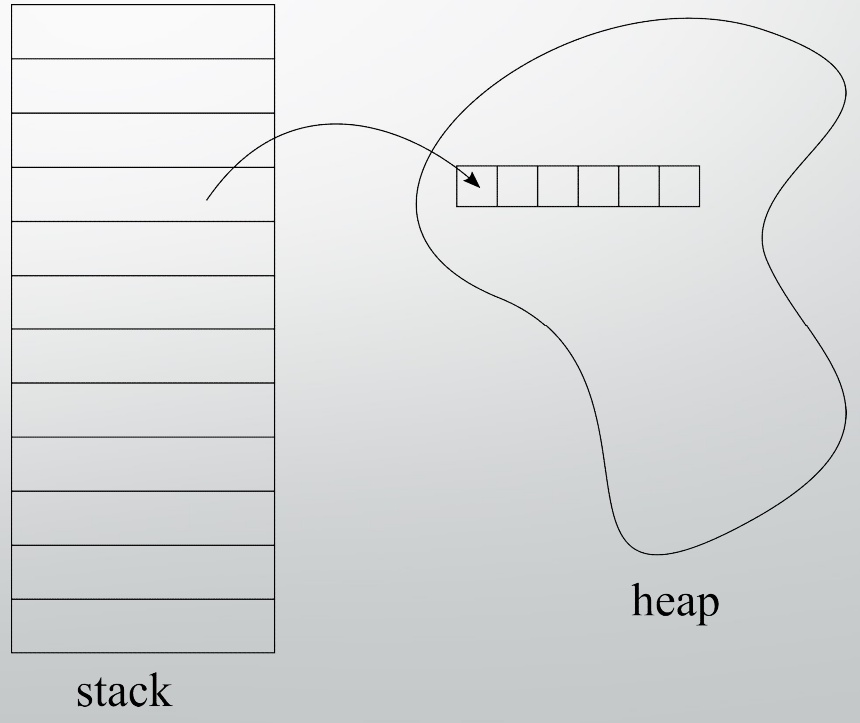
Conceitos Básicos

- Blocos de código → Delimita regiões
- Léxico → Escopo de variável

Memória

- Tipos de Memória
  - *Stack*
  - *Heap*

- *Stack* → Memória de tipos primitivos e endereços
- *Heap* → Memória de objetos



Memória

- Saídas Esperadas?

```
int main(void)
{
	int a = 0;
	int b = 1;

	b = a;

	cout << b << "\n";

	//Pause
	system("PAUSE");

	return 0;
}
```

b = 0

```
int main(void)
{
	int a = 0;
	int b = 1;

	b = a;

	a = 10;

	cout << b << "\n";

	//Pause
	system("PAUSE");

	return 0;
}
```

b = 0

```
int main(void)
{
	int a = 0;
	int b = 0;

	foo(b);

	cout << b << "\n";

	//Pause
	system("PAUSE");

	return 0;
}

void foo(int a)
{
	a = 2;
}
```

b = 0

Memória

- Respostas esperadas?

```
int main(void)
{
	int* a = new int[3];
	int* b = new int[3];

	for (int i = 0; i < 3; i++)
	{
		a[i] = i + 1;
		b[i] = i + 4;
	}

	for (int i = 0; i < 3; i++)
	{
		cout << a[i] << "\t";
	}

	cout << "\n";

	b = a;

	for (int i = 0; i < 3; i++)
	{
		cout << b[i] << "\t";
	}

	cout << "\n";

	//Pause
	system("PAUSE");

	return 0;
}
```

b={1,2,3}

```
int main(void)
{
	int* a = new int[3];
	int* b = new int[3];

	for (int i = 0; i < 3; i++)
	{
		a[i] = i + 1;
		b[i] = i + 4;
	}

	for (int i = 0; i < 3; i++)
	{
		cout << a[i] << "\t";
	}

	cout << "\n";

	b = a;

	foo(a);

	for (int i = 0; i < 3; i++)
	{
		cout << b[i] << "\t";
	}

	cout << "\n";

	//Pause
	system("PAUSE");

	return 0;
}

void foo(int* a)
{
	a[2] = 10;
}
```

b={1,2,10}

Objetos e Classes

- Exemplo de uma classe *Carro*.

```
public class Carro
{
        public int Portas { get; set; }

        public int Rodas { get; set; }

        public String Motor { get; set; }

        public String Direcao { get; set; }

        public bool TemABS { get; set; }

        public bool TemTrava { get; set; }

        public bool TemVidro { get; set; }

        public double VelocidadeMaxima { get; set; }

        public double Velocidade { get; set; }

        public void Acelerar(double aceleracao) { }

        public void Freiar(double desaceleracao) { }

 }
```

Objetos e Classes

- Métodos:

```
public void Acelerar(double aceleracao)
{
     if (Velocidade < VelocidadeMaxima)
     {
         Velocidade = Velocidade + aceleracao * 0.1;
         if(Velocidade > VelocidadeMaxima)
              Velocidade = VelocidadeMaxima
     }
}

public void Freiar(double desaceleracao)
{
     if (Velocidade > 0)
     {
          Velocidade = Velocidade - desaceleracao * 0.1;
          if (Velocidade < 0)
                Velocidade = 0;
      }
}
```

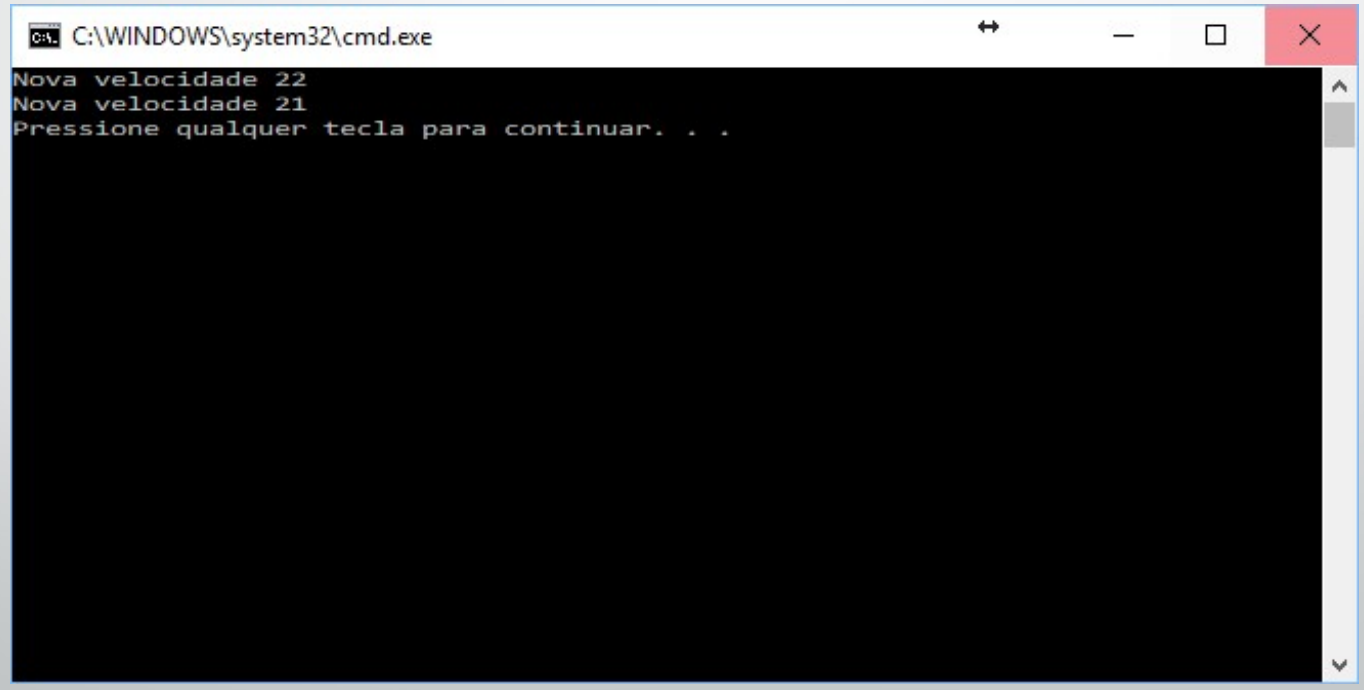
Objetos e Classes

- Uso e inicialização
  - Objetos!

```
static void Main(string[] args)
{
      Carro fusca = new Carro();

      fusca.Direcao = "tradicional";
      fusca.Portas = 2;
      fusca.Rodas = 4;
      fusca.Motor = "1.0";
      fusca.TemABS = false;
      fusca.TemTrava = false;
      fusca.TemVidro = false;
      fusca.VelocidadeMaxima = 100;
      fusca.Velocidade = 20;

      fusca.Acelerar(20);

      Console.WriteLine("Nova velocidade {0}", fusca.Velocidade);

      fusca.Freiar(10);

      Console.WriteLine("Nova velocidade {0}", fusca.Velocidade);
}
```

Objetos e Classes

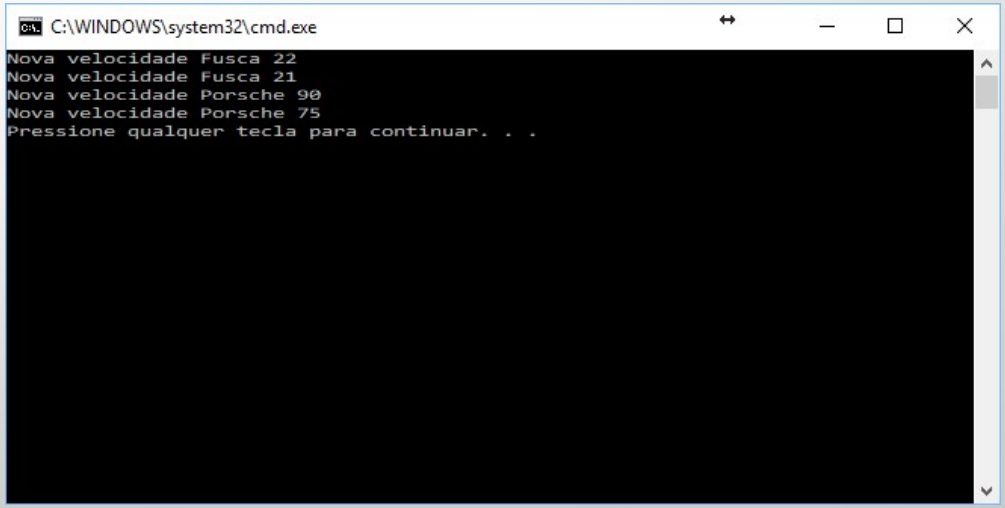
- Saída:



Objetos e Classes

- Uso com 2 objetos:

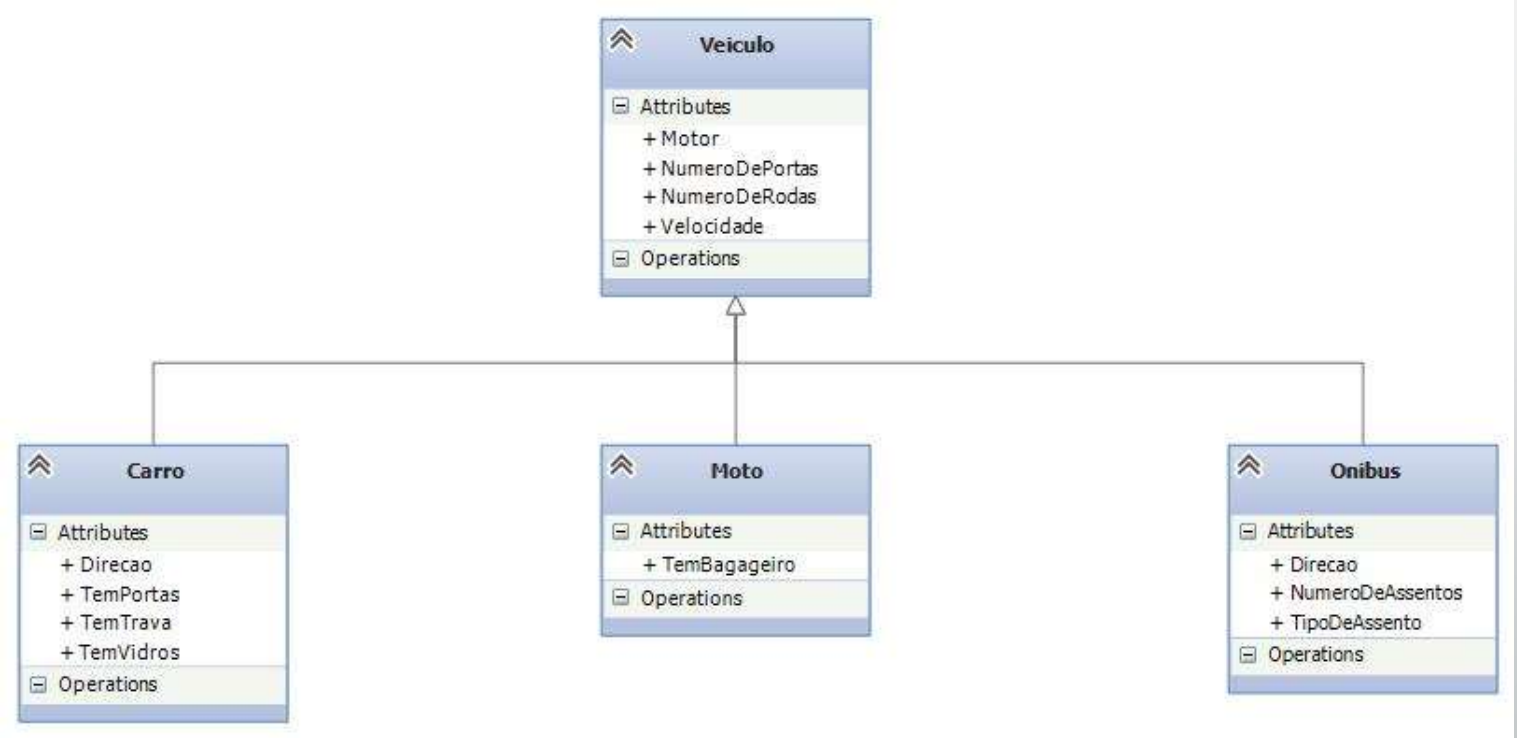
```
static void Main(string[] args)
      {
            Carro fusca = new Carro();

            fusca.Direcao = "tradicional";
            fusca.Portas = 2;
            fusca.Rodas = 4;
            fusca.Motor = "1.0";
            fusca.TemABS = false;
            fusca.TemTrava = false;
            fusca.TemVidro = false;
            fusca.VelocidadeMaxima = 100;
            fusca.Velocidade = 20;

            fusca.Acelerar(20);

            Console.WriteLine("Nova velocidade Fusca {0}", fusca.Velocidade);

            fusca.Freiar(10);

            Console.WriteLine("Nova velocidade Fusca {0}", fusca.Velocidade);

            Carro porsche = new Carro();

            porsche.Direcao = "eletrica";
            porsche.Portas = 2;
            porsche.Rodas = 4;
            porsche.Motor = "2.0";
            porsche.TemABS = true;
            porsche.TemTrava = true;
            porsche.TemVidro = true;
            porsche.VelocidadeMaxima = 200;
            porsche.Velocidade = 80;

            porsche.Acelerar(100);

            Console.WriteLine("Nova velocidade Porsche {0}", porsche.Velocidade);

            porsche.Freiar(150);

            Console.WriteLine("Nova velocidade Porsche {0}", porsche.Velocidade);
      }
```



Herança

- A ideia de herança é poder definir uma estrutura comum de uma classe e definir classes que possuam essas características.
- Em geral, pode-se dizer que se utilizam palavras chave como *virtual* e *abstract.*
- A ideia é criar classes virtuais ou abstratas que definam essas características (denominadas classes mãe) e depois criar classes que estendam as funcionalidades (classes filhas).

Herança


Veiculo
Attributes
+ Motor
+ NumeroDePortas
+ NumeroDeRodas
+ Velocidade
Operations
Carro
Attributes
+ Direcao
+ TemPortas
+ TemTrava
+ TemVidros
Operations
Moto
Attributes
+ TemBagageiro
Operations
Onibus
Attributes
+ Direcao
+ NumeroDeAssentos
+ TipoDeAssento
Operations

Herança

- Isso permite utilizar, por exemplo, para a criação do 'estacionamento'.

```
public static void Main(string[] args)
{
    Veiculo[] estacionamento = new Veiculo[10];

    estacionamento[0] = new Carro();
    estacionamento[1] = new Carro();
    estacionamento[2] = new Moto();
    estacionamento[3] = new Caminhao();
    estacionamento[4] = new Carro();
}
```

Polimorfismo

- Isso permite definir funções com mesmo nome, porém com assinatura diferente (mesmo número de parâmetros com tipos diferentes ou número diferentes de parâmetros).
- Útil na hora de definir funções com parâmetros padronizados ou que mudam o comportamento de acordo com a entrada.
- Exemplos:
  - *solver* de problemas lineares: pode ter como entrada uma matriz e um vetor ou duas matrizes. As respostas são um vetor ou uma matriz tal que resolva o sistema linear para cada uma das entradas.

Polimorfismo

- *Overloading*: permite sobrepor uma função da classe mãe na classe filha.
  - Útil para sobrepor os operadores de uma classe. São eles: +, -, x, /, %, entre outros.
- Exemplo: classe de matrizes (ou vetores) que redefinem soma, subtração multiplicação e divisão de matrizes por matrizes, matrizes por vetor e matrizes por número.
- Dessa forma, quando se utilizar uma matriz em um código e somar ela com outra, o resultado será computado por esse operador modificado

Usos

- Construtor: permite criar objetos com parâmetros diferentes e que alocam variáveis de maneira diferente de acordo com a chamada.

- *Operator Overloading*: permite a definição / sobreposição de operações para determinadas classes.