Fundamentos de C#

Classes e Estruturas

Tópicos Abordados

- Classes e objetos
- Fields
- Métodos
- Armazenamento em memória
  - Stack
  - Managed heap
- Passagem de parâmetros
  - Por valor
  - Por referência
  - Parâmetos de saída
  - Parâmetros opcionais
  - O modificador *params*

Tópicos Abordados

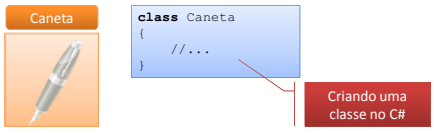
- Sobrecarga de métodos
- Nullable Types
- Operador *??* (null-coalescing)
- Estruturas
  - Instanciação
  - Diferenças entre classes e estruturas

Orientação a Objetos

- Durante muito tempo a programação procedural foi dominante
  - Ex: Pascal, COBOL, C, Visual Basic 6
  - É um paradigma centrado em procedimentos
- Linguagens de programação mais modernas passaram a adotar a programação orientada a objetos
  - Ex: C++, Java, C#, Visual Basic .NET
  - É um paradigma de programação centrado em objetos e na interação entre eles

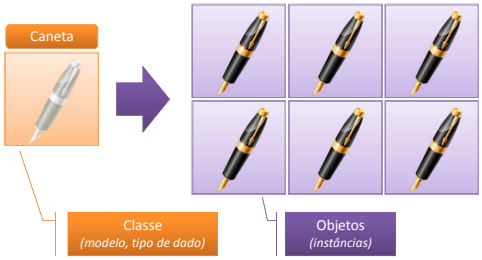
Classes

- Desempenham um papel central na orientação a objetos
- Definem um **modelo** para os objetos
- Classes são **tipos de dados**



Objetos

- A partir de uma classe, podemos criar **objetos** (ou **instâncias**)



Criação de Objetos

- A criação (instanciação) de um objeto é feita usando o operador **new**

```
Caneta c1 = new Caneta();
Caneta c2 = new Caneta();
```

Criação de dois objetos da classe **Caneta**

Fields

- Uma classe pode definir **fields**
- Representam características, informações, atributos objetos de uma classe

Características da caneta *(substantivos)*

```
class Caneta
{
    int cor;
    string marca;
    int anoFabricacao;
    int corTinta;
}
```

Fields

Fields

- Os *fields* podem ser chamados diretamente em objetos
  - Dependendo da sua visibilidade

```
class Caneta
{
    public int cor;
    public string marca;
    public int anoFabricacao;
    public int corTinta;
}
```

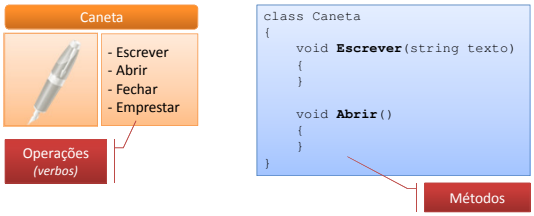
Visibilidade **public**

```
Caneta c = new Caneta();
c.cor = 10;
c.marca = "Bic";
Console.WriteLine(c.marca);
```

O acesso é feito através do "."

Métodos

- Métodos são operações associadas à classe
- Estas operações podem ser invocadas por alguém que quer interagir com objetos da classe

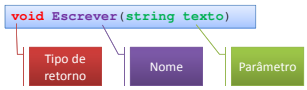


Declaração de Métodos

- Um método tem uma assinatura

```
[modificadores] tipo_retorno Nome([parâmetros])
```

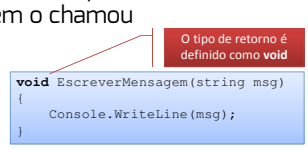
- Exemplo



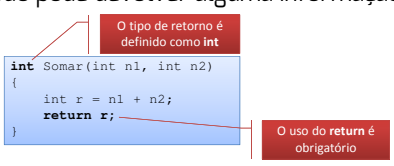
- Método **Main()**

Retorno em Métodos

- Um método não precisa devolver uma informação para quem o chamou



- Um método pode devolver alguma informação



Invocação de Métodos

- Métodos podem ser invocados em objetos
  - Depende da visibilidade

```
class Math
{
    public int Somar(int n1, int n2)
    {
        return n1 + n2;
    }
}
```

Visibilidade **public**

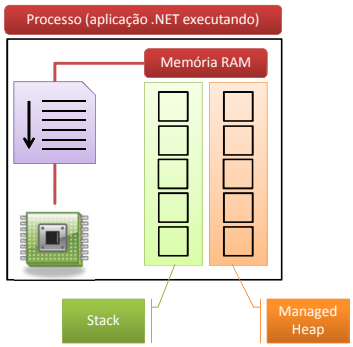
```
Math m = new Math();
int soma = m.Somar(15, 6);
```

O acesso é feito através do "."

Passagem de Parâmetros

- Métodos podem receber zero ou mais parâmetros
- Para entender como funciona a passagem de parâmetros, é preciso entender como os dados da aplicação são armazenados na memória

Execução de uma Aplicação .NET



Value Types e Reference Types

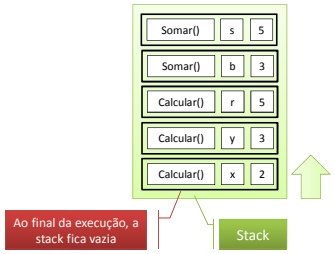
- Os tipos de dados em C# são divididos em duas categorias
  - Value Types
    - *byte*, *short*, *int*, *long*, *float*, *double*, *decimal*, etc.
  - Reference Types
    - Classes
- Existe uma diferença importante entre essas duas categorias de tipos

Funcionamento da Stack

- Stack significa **pilha**
  - Empilha **variáveis locais** e **parâmetros** passados para métodos

```
void Calcular()
{
    int x = 2;
    int y = 3;
    int r = Somar(x, y);
}
```

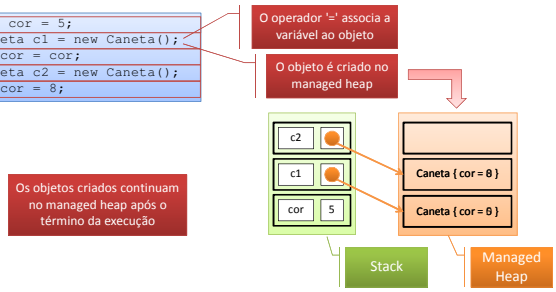
```
void Somar(int a, int b)
{
    int s = a + b;
    return s;
}
```



Funcionamento do Managed Heap

- O managed heap é o local onde são criados os objetos

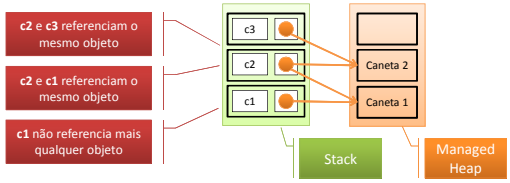
```
int cor = 5;
Caneta c1 = new Caneta();
c1.cor = cor;
Caneta c2 = new Caneta();
c2.cor = 8;
```



Funcionamento do Managed Heap

```
Caneta c1 = new Caneta();
Caneta c2 = new Caneta();
Caneta c3 = c2;
c2 = c1;
c1 = null;
```

Os objetos criados continuam no managed heap após o término da execução



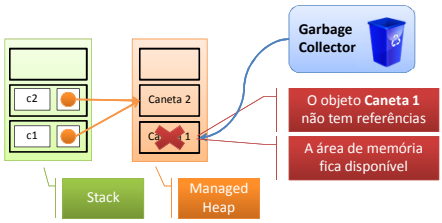
Garbage Collector

- É um serviço fornecido pela plataforma .NET através do CLR
- Faz uma limpeza no managed heap
  - Remove objetos que não têm mais referências
- O CLR decide quando o garbage collector vai executar
  - É possível forçar a execução do garbage collector via programação
    - Deve ser usado apenas para fins de teste

```
GC.Collect();
```

Executa o garbage collector

Garbage Collector

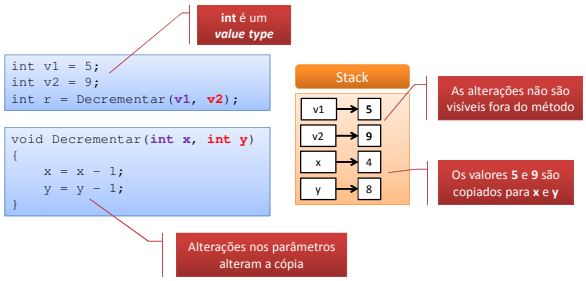


Tipos de Passagem de Parâmetros

- Parâmetros podem ser fornecidos a métodos de duas formas
  - Por valor
    - O valor é copiado para o parâmetro
  - Por referência
    - A referência à área de memória onde o dado está armazenado é fornecida como parâmetro
- No C#, o padrão é passar os parâmetro por valor

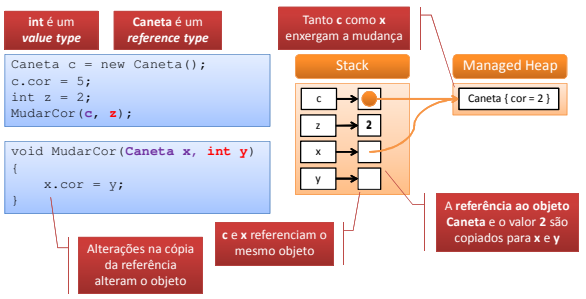
Parâmetros por Valor

- O parâmetro recebe uma cópia do valor que está sendo fornecido



Parâmetros por Valor

- Para *reference types*, funciona da mesma forma, mas o resultado final é diferente



Parâmetros por Referência

- Na passagem por referência a área de memória é passada como parâmetro
- É preciso usar o modificador **ref** na declaração do método e na chamada

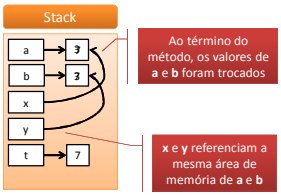
```
void Trocar(ref int x, ref int y)
{
    int t = x;
    x = y;
    y = t;
}
```

```
int a = 7;
int b = 3;
Trocar(ref a, ref b);
```

Parâmetros por Referência

```
int a = 7;
int b = 3;
Trocar(ref a, ref b);
```

```
void Trocar(ref int x, ref int y)
{
    int t = x;
    x = y;
    y = t;
}
```



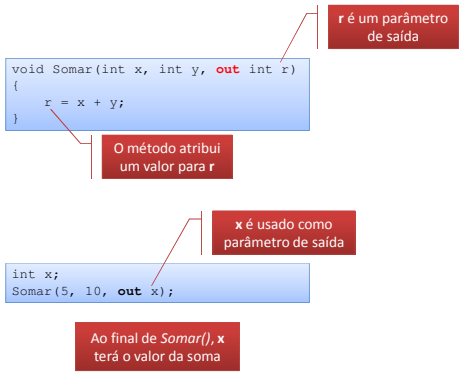
Ao término do método, os valores de **a** e **b** foram trocados

x e **y** referenciam a mesma área de memória de **a** e **b**

Parâmetros de Saída

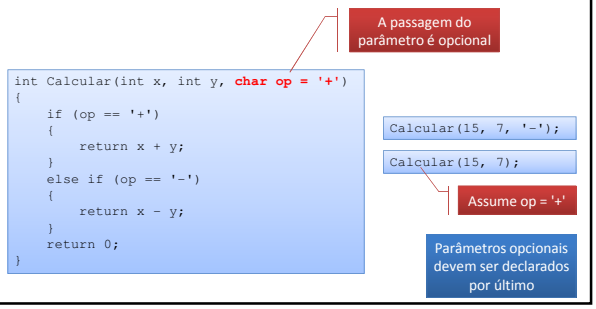
- São parâmetros cujos valores são definidos pelo método chamado
  - Quem chama o método não precisa inicializar a variável
- Os parâmetros de saída são designados através do modificador **out**
  - O C# passa parâmetros de saída como referência de forma automática

Parâmetros de Saída



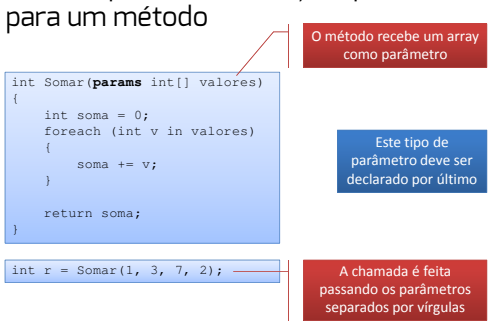
Parâmetros Opcionais

- São parâmetros que podem ou não ser passados
  - Se não forem passados, assumem um valor padrão



O modificador *params*

- Permite passar um array de parâmetros para um método



Sobrecarga de Métodos

- Sobrecarregar um método significa criar outros métodos com o mesmo nome, mas com assinatura diferente

`int Somar(int a, int b)`	(int, int) -> int
`int Somar(int a, int b, int c)`	(int, int, int) -> int
`double Somar(double a, double b)`	(double, double) -> double
`long Somar(long a, long b, long c)`	(long, long, long) -> int
`void Somar(int a, int b, out int c)`	(int, int, out int) -> void

A mudança pode ser feita nos parâmetros (tipo e/ou quantidade) e no tipo de retorno

Mudar só o tipo do retorno gera erro de compilação

Sobrecarga de Métodos

- Exemplos

```
Somar(9, 9);          → int Somar(int a, int b)
Somar(5, 3, 7);       → int Somar(int a, int b, int c)
Somar(2.3, 1.2);      → double Somar(double a, double b)
Somar(1.1, 6);        → double Somar(double a, double b)
Somar(4L, 3, 3);      → long Somar(long a, long b, long c)
```

Nullable Types

- Permite que *value types* possam receber **null**
  - Não pode ser usado por *reference types*

```
bool b = null;
int i = null;
```

Não compila, pois *null* só pode ser usado com *reference types*

```
bool? b = null;
int? i = null;
```

O uso do **?** como sufixo do value type permite a atribuição do *null*

```
if (b != null) {}

if (b.HasValue) {}
```

Permite testar se o valor da variável é *null*

Operador *??*

- *Null-coalescing operator*
- Permite atribuir um valor padrão a uma variável se o valor dela for *null*

```
bool? b1 = null;
bool b2 = b1 ?? false;
```

b2 recebe o valor de **b1**. Se **b1** for **null**, **b2** recebe **false**

```
bool? b1 = null;
bool b2;
if (b1 != null)
{
    b2 = b1;
}
else
{
    b2 = false;
}
```

Sem o uso do operador *??*, o código fica maior

O operador *??* também pode ser usado com *reference types*

Estruturas

- Também chamadas de *strutctures* ou *structs*
- São estruturas de dados usadas para agrupar dados
- A declaração é bastante similar às classes
  - Estruturas também podem conter fields e/ou métodos

```
struct Fracao
{
    public double numerador;
    public double denominador;

    public double Dividir()
    {
        return numerador / denominador;
    }
}
```

A palavra-chave **struct** é utilizada

Fields

Métodos

Instanciação de Estruturas

- A instanciação é similar a encontrada em classes

```
Fracao f = new Fracao();
```

Uso do operador *new()*

- Pode ser feita de outra forma

```
Fracao f;
f.numerador = 5;
f.denominador = 10;
```

Todas as variáveis de instância devem ser inicializadas

Diferenças entre Classes e Estruturas

	Classe	Estrutura
Declaração	Palavra-chave *class*	Palavra-chave *struct*
Categoria do tipo	Reference type	Value type
Local de armazenamento na memória	Managed Heap	Stack
Construtores	Pode ter um construtor sem parâmetros	Sempre tem um construtor sem parâmetros
Herança	Suporta	Não suporta

- Estruturas devem ser usadas para tipos de dados simples
  - O fato de estruturas serem *value types* impacta na passagem de parâmetros
  - Por outro lado, são rapidamente removidas da memória, pois são criadas na stack

Nested Types

- Um *nested type* (ou *inner type*) é um tipo definido dentro de uma classe ou estrutura

```
class Outer
{
    class Inner
    {
    }
}
```

Nested type

Por padrão, todo *nested type* é *private*, mas a visibilidade pode ser alterada

- O tipo externo tem acesso a tudo definido no tipo interno

Nested Types

- O nome de um *nested type* é composto também pelo tipo externo

```
class Outer
{
    public class Inner
    {
    }
}
```

```
Inner i = new Inner();
```

Erro de compilação

```
Outer.Inner i = new Outer.Inner();
```

OK

A Notação UML

- **U**nified **M**odeling **L**anguage
- Utilizada para documentar sistemas orientados a objetos
- Composta por diversos diagramas
  - Um deles é o **Diagrama de Classes**, que mostra as classes do sistema, juntamente com seus respectivos métodos e fields

Diagrama de Classes

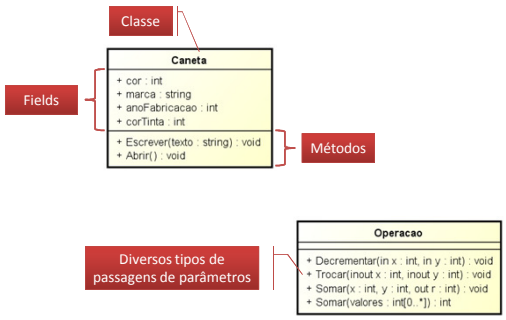


Diagrama de Classes