PEF – 3528 – Ferramentas Computacionais na Mecânica das Estruturas: Criação e Concepção

Prof. Dr. Rodrigo Provasi

e-mail: provasi@usp.br

Sala 09 – LEM – Prédio de Engenharia Civil

Introdução ao C#

Parte II – Classes

Classes

- O conceito de classes está associado ao conceito de orientação à objetos que existe em algumas linguagens (C#, C++, Java, Python, entre outras).
- A ideia por trás é poder criar classificações.
- Com as classes é possível definir característica comuns à um grupo de objetos e utilizar essa ideia na geração de um código mais complexo.

Classes

- A criação de classes tem por trás um princípio importante: o *encapsulamento*.
- Com isso, pode-se usar classes sem saber exatamente os detalhes da implementação.
- Exemplo, utiliza-se o método *Console.WriteLine()* sem saber como ele funciona internamente, ou seja, o seu conteúdo está encapsulado na classe *Console*.

Classes

- Para se definir uma classe no C#, utiliza-se a palavra chave *class*. Exemplo:

```
class Circle
{
    int radius;
    double Area()
    {
        return Math.PI * radius * radius;
    }
}
```

Classes

- Classes são compostas por métodos e variáveis (*fields*).
- No caso da classe *Circle*, tem-se:

  int radius; ← *variável*

  double Area() ← *método*

Classes

- Uma vez definida uma classe, pode-se instanciar um objeto dessa classe:

```
Circle c;                 // Create a Circle variable
c = new Circle();         // Initialize it
```

Classes

- Importante!! Para criar uma variável de um tipo não primitivo, precisa-se utilizar a palavra chave *new*.

- Porém, é possível atribuir uma variável a outra já existente:

```
Circle c;
c = new Circle();
Circle d;
d = c;
```

Controlando acesso

- A classe anteriormente exibida encapsula os dados, criando uma 'barreira' para o exterior.
- As variáveis podem ser vistas na classe porém não fora dela → Escopo
- Apesar de se poder criar uma variável da classe *Circle*, não é possível acessar o raio!

Controlando o acesso

```
class Circle
{
        private int radius;
        public double Area()
        {
                return Math.PI * radius * radius;
        }
}
```

Controlando o acesso

- *public* : todos podem ver e acessar a variável / método.
- *private* : somente a classe pode ver e acessar a variável / método.
- *protected* : a classe e as filhas* podem ver e acessar a variável / método.

- Por padrão, métodos e variáveis que não têm o modificador explícito são *private*.

* isso será explicado melhor na parte de herança e polimorfismo.

Construtor

- Toda vez que uma variável é instanciada (usando o *new*), chama-se o construtor da classe.
- Esse é um método especial que tem o mesmo nome da classe e que pode ou não ter parâmetros.

Construtor

```
class Circle
{
        private int radius;
        public Circle() // default constructor
        {
                radius = 0;
        }
        public double Area()
        {
                return Math.PI * radius * radius;
        }
}
```

Construtor

- Agora é possível executar o seguinte código:

```
Circle c;
c = new Circle();
double areaOfCircle = c.Area();
```

Construtor

- No caso anterior, o resultado da área é zero e sempre será porque não há meios de modificar o valor do raio (mais sobre isso no tópico *properties*).
- Para contornar isso, pode-se definir um segundo construtor com parâmetros*.

* Será visto com detalhes em *overloading* de métodos.

Construtor

```
class Circle
{
        private int radius;
        public Circle() // default constructor
        {
                radius = 0;
        }
        public Circle(int initialRadius) // overloaded constructor
        {
                radius = initialRadius;
        }
        public double Area()
        {
                return Math.PI * radius * radius;
        }
}
```

Constutor

- Assim, pode-se criar um novo objeto do tipo *Circle* como:

Circle c;

c = new Circle(45);

Palavra chave *static*

- Métodos e variáveis marcados com a palavra *static* são da classe e não da instância. Por exemplo, o *PI* da classe *Math* (usado na classe *Circle*) é uma variável estática.

Palavra chave *static*

- Um outro exemplo é o método *Sqrt* da classe *Math*. A declaração é:

```
class Math
{
    public static double Sqrt(double d)
    {
        ...
    }
    ...
}
```

Palavra chave *static*

```
class Circle
{
        private int radius;
        public static int NumCircles = 0;
        public Circle() // default constructor
        {
                radius = 0;
                NumCircles++;
         }
        public Circle(int initialRadius) // overloaded constructor
        {
                radius = initialRadius;
                 NumCircles++;
        }
}
```

Palavra chave *static*

- No exemplo anterior, a variável *NumCircles* é estática, ou seja, ela não pertence a um objeto, mas sim à Classe.
- Toda vez que uma classe é instanciada, as variáveis são atreladas a ela, exceto *NumCircles*, que pertence a classe.
- No exemplo, todo círculo instanciado, aumenta em 1 o valor de *NumCircles*.

Palavra chave *static*

- Já para criar uma variável *static* usa-se a seguinte sintaxe (exemplo *PI* na classe *Math*):

```
class Math
{
    ...
    public const double PI = 3.14159265358979323846;
}
```

Palavra chave *static*

- Além disso, classes podem ser declaradas como estáticas. Numa classe estática, apenas membros estáticos podem existir.

- Um possível uso é para a definição de *extension methods*.

Classes

- Há ainda um tipo de classe denominado Anônima, cuja criação utiliza a palavra chave *var* e é criada *inline* (ou seja, diretamente do código). Esse tipo não será tratado aqui.

- Outra possibilidade é o que se chama de *nested class*, cuja definição da classe está em outra classe e, por causa dos modificadores de acesso, pode ser apenas acessada por essa classe externa.

Introdução ao C#

Parte II – Memória

Valor versus referência

- Tipos como *int*, *float*, *double*, *char* (mas não *string*) são genericamente chamados de tipos valor. Esses tipos tem um valor fixo de memória e são previamente alocados pelo compilador.
- Já classes, *strings* e vetores / matrizes são chamados de tipos referência. Nesse caso o compilador reserva uma parte da memória para guardar o endereço que estará contido o objeto.

Tipo Valor

- Observe o seguinte exemplo:

```
int i = 42;          // declare and initialize i
int copyi = i;       /* copyi contains a copy of the data in i: i and copyi both
                        contain the value 42 */
i++;                 /* incrementing i has no effect on copyi; i now contains 43, but
                        copyi still contains 42 */
```

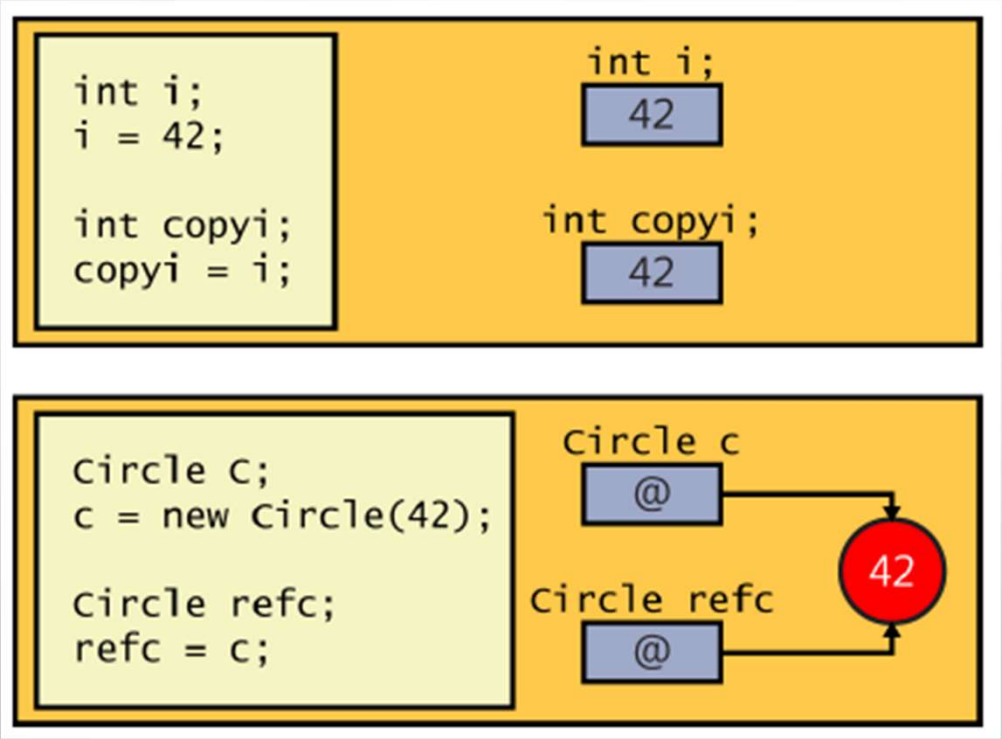
Tipo Referência

- Agora observe o seguinte exemplo:

Circle c = new Circle(42);

Circle refc = c;

Valor versus referência



Referência

- Para copiar objetos que são do tipo referência não deve se fazer o seguinte:

```
Circle refc = new Circle();
refc.radius = c.radius; // Don't try this
```

- O correto é dar um *override* na função *Clone* e chama-la ao invés de copiar o objeto.

Clone

```
class Circle
{
        private int radius; // Constructors and other methods omitted
        ...
        public Circle Clone()
        {
                // Create a new Circle object
                Circle clone = new Circle();
                // Copy private data from this to clone
                clone.radius = this.radius;
                // Return the new Circle object containing the copied data
                return clone;
        }
}
```

Null e tipos *nullable*

- Quando se declara uma variável, a melhor coisa a se fazer é inicializá-la:

```
int i = 0;
double d = 0.0;
```

- Se for uma classe, por exemplo:

```
Circle c = new Circle(42);
Circle copy = new Circle(99); // Some random value, for initializing copy...
copy = c; // copy and c refer to the same object
```

Null e tipos *nullable*

- Mas e se só se desejar fazer a copia do círculo ser o *c* se *copy* não tiver valor prévio?

```
Circle c = new Circle(42);
Circle copy; // Uninitialized !!!...
if (copy == // only assign to copy if it is uninitialized, but what goes here?)
{
    copy = c; // copy and c refer to the same object
    ...
}
```

Null

- Para tal pode-se utilizar a palavra chave *null:*

```
Circle c = new Circle(42);
Circle copy = null; // Initialized…
if (copy == null)
{
    copy = c; // copy and c refer to the same object
    …
}
```

Tipos *nullable*

- O seguinte trecho de código é ilegal:

int i = null; // illegal

- Como fazer para *settar* o valor do inteiro para null?
- O C# permite isso usando um tipo *nullable*:

int? i = null; // legal

Tipos *nullable*

- Agora é possível escrever a expressão:

if (i == null) …

- Observando o código a seguir vê-se algumas coisas que são permitidas ou não fazer com tipos *nullable:*

```
int? i = null;
int j = 99;
i = 100; // Copy a value type constant to a nullable type
i = j; // Copy a value type variable to a nullable type
j = i; // Illegal
```

Tipos *nullable*

- Esses tipos possuem duas propriedades muito importantes:

```
int? i = null;
...
if (!i.HasValue)
{
        // If i is null, then assign it the value 99
        i = 99;
}
else
{
        // If i is not null, then display its value
        Console.WriteLine(i.Value);
}
```

Usando *ref* e *out*

- Imagine o seguinte código:

```
static void doIncrement(int param)
{
        param++;
}
static void Main()
{
        int arg = 42;
        doIncrement(arg);
         Console.WriteLine(arg); // writes 42, not 43
}
```

Usando ref e out

- O que houve no código anterior?
- *int* é tipo de referência e, quando passado para a função, uma cópia é criada.
- Para alterar um tipo valor em uma função, é preciso passar a variável por referência.
- Para tal pode-se usar duas palavras chaves: *ref* e *int*.

Usando *ref*

- Para o problema anterior, pode-se passar por referência o inteiro:

```
static void doIncrement(ref int param) // using ref
{
        param++;
}
static void Main()
{
        int arg = 42;
        doIncrement(ref arg); // using ref
        Console.WriteLine(arg); // writes 43
}
```

Usando *ref*

- Agora, se o inteiro não for inicializado? O seguinte código produz um erro:

```
static void doIncrement(ref int param)
{
        param++;
}
static void Main()
{
        int arg; // not initialized
        doIncrement(ref arg);
        Console.WriteLine(arg);
}
```

Usando *out*

- Para tal, utiliza-se *out*, porém a variável tem que ser inicializada no corpo da função:

```
static void doInitialize(out int param)
{
    param = 42; // Initialize param before finishing
}
```

- O Código a seguir não compila, porque não inicializa a variável:

```
static void doInitialize(out int param)
{
    // Do nothing
}
```

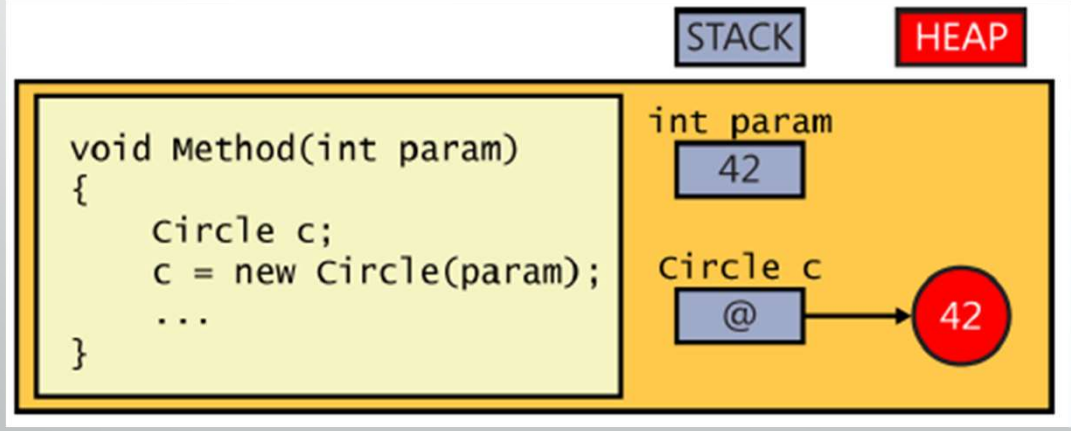
Usando *out*

- Sendo assim, o código a seguir compila e funciona:

```
static void doInitialize(out int param)
{
        param = 42;
}
static void Main()
{
        int arg; // not initialized
        doInitialize(out arg); // legal
        Console.WriteLine(arg); // writes 42
}
```

Memória

- De uma maneira geral, pode-se dizer que as variáveis tipo e endereços ficam armazenadas em uma pilha denominada *stack* e o demais dados ficam na memória geral denominada *heap*.



Classe *System.Object*

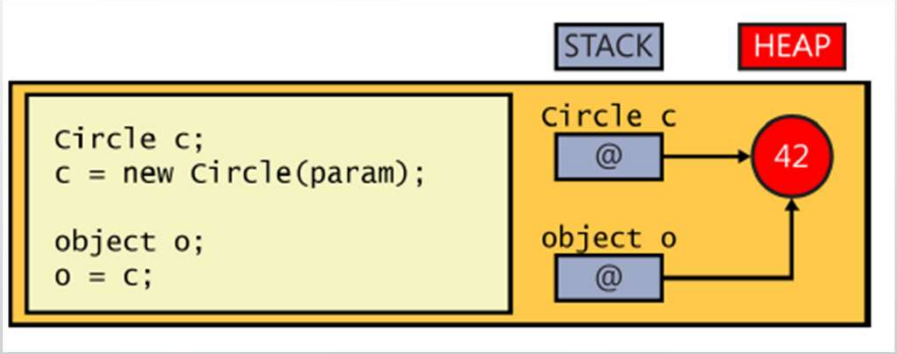
- A classe *System.Object* (que pode ser inicializada como o tipo *object*) é a classe da qual deriva todas as classes do C#. Ela funciona como uma classe genérica que aceita qualquer tipo de objeto. Por exemplo:

Circle c;

c = new Circle(42);

object o;

o = c;

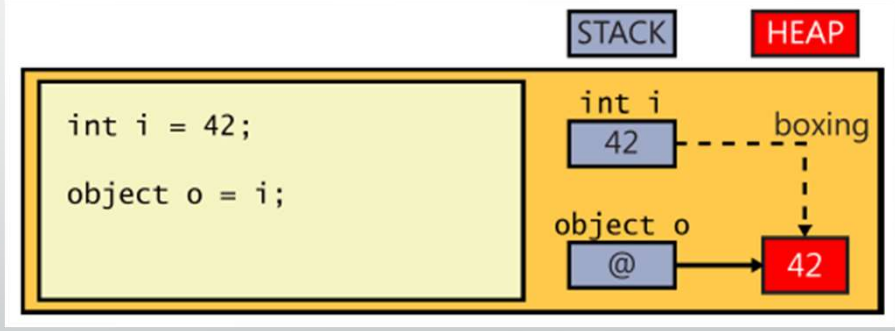


Boxing

- Como foi visto, a classe *object* pode abrigar qualquer objeto do C#. Então é possível escrever:

int i = 42;

object o = i;



Unboxing

- Por outro lado, e se fizermos a operação inversa? Essa operação se chama *casting*.

```
Circle c = new Circle();
int i = 42;
object o;
o = c; // o refers to a circle
i = o; // what is stored in i?
```

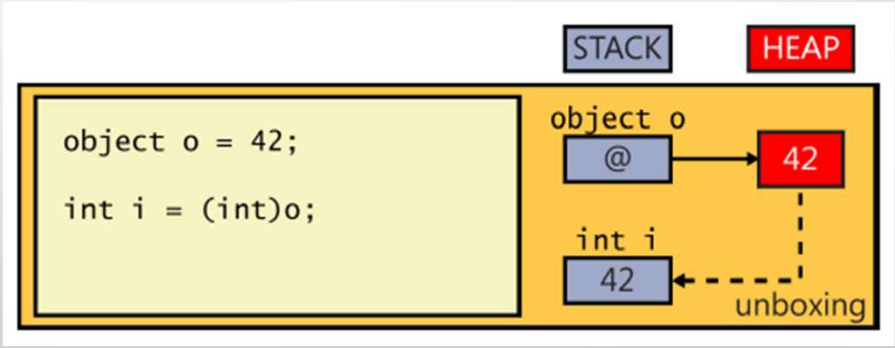
Unboxing

- Para resolver o problema necessita-se explicitar para o que será transformado a variável:

int i = 42;

object o = i; // boxes

i = (int)o; // compiles okay



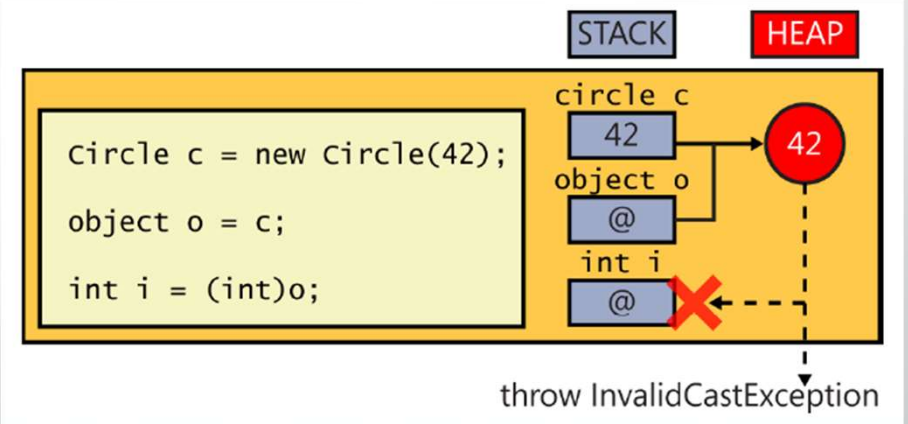
Unboxing

- Porém, se o formato da variável destino for incompatível uma exceção do tipo *InvalidCastException*, apesar de compilar sem problemas.

Circle c = new Circle(42);

object o = c; // doesn't box because Circle is a reference variable

int i = (int)o; // compiles okay but throws an exception at run time



Operador *is*

- Como fazer para que a operação de *casting* seja feita corretamente?
- Para tal, pode-se se utilizar o operador *is:*

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
if (o is WrappedInt)
{
    WrappedInt temp = (WrappedInt)o;  //This is safe; o is a WrappedInt
    ...
}
```

Operador *as*

- Existe um outro operador que faz algo similar porém facilita a operação de casting. Ele é o *as:*

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
WrappedInt temp = o as WrappedInt;
if (temp != null)
{
        ... // Cast was successful
}
```

Introdução ao C#

Parte II – Enumerações e Estruturas

Enumerações e Estruturas

- São tipos valor → Armazenados no *stack* e tratado tal como um *int*.
- Enumerações funcionam como rótulos para uma sequencia numérica e servem para facilitar comparações.
- Estruturas são similares as classes, porém com algumas restrições.

Enumerações

- A sintaxe para declarar uma enumeração é bem simples:

```
enum Season { Spring, Summer, Fall, Winter }
class Example
{
        public void Method(Season parameter) // method parameter example
        {
                Season localVariable; // local variable example
                ...
        }
        private Season currentSeason; // field example
}
```

Enumerações

- Se for impressa em tela:

```
Season colorful = Season.Fall;
Console.WriteLine(colorful); // writes out 'Fall'
```

- E se for convertida para *string*:

```
string name = colorful.ToString();
Console.WriteLine(name); // also writes out 'Fall'
```

Enumerações

```
enum Season { Spring, Summer, Fall, Winter }
...
Season colorful = Season.Fall;
Console.WriteLine((int)colorful); // writes out '2'
```

```
enum Season { Spring = 1, Summer, Fall, Winter }
```

Estruturas

- Estruturas podem ter métodos, propriedades e construtores. A sintaxe é:

```
struct Time
{
        private int hours, minutes, seconds;
        ...
        public Time(int hh, int mm, int ss)
        {
                this.hours = hh % 24;
                this.minutes = mm % 60;
                this.seconds = ss % 60;
        }
        public int Hours()
        {
                return this.hours;
        }
}
```

Estruturas

- Não pode se declarar um construtor que não tenha parâmetros em uma estrutura.

```
struct Time
{
        private int hours, minutes, seconds;
        ...
        public Time(int hh, int mm)
        {
                this.hours = hh;
                this.minutes = mm;
        } // compile-time error: seconds not initialized
}
```

Estruturas

Question	Structure	Class
Is this a value type or a reference type?	A structure is a value type.	A class is a reference type.
Do instances live on the stack or the heap?	Structure instances are called values and live on the stack.	Class instances are called objects and live on the heap.
Can you declare a default constructor?	No.	Yes.
If you declare your own constructor, will the compiler still generate the default constructor?	Yes.	No.
If you don't initialize a field in your own constructor, will the compiler automatically initialize it for you?	No.	Yes.
Are you allowed to initialize instance fields at their point of declaration?	No.	Yes.

Estruturas

```
struct Time
{
        private int hours, minutes, seconds;
        ...
}

class Example
{
        private Time currentTime;

        public void Method(Time parameter)
        {
                Time localVariable;
                ...
        }
}
```

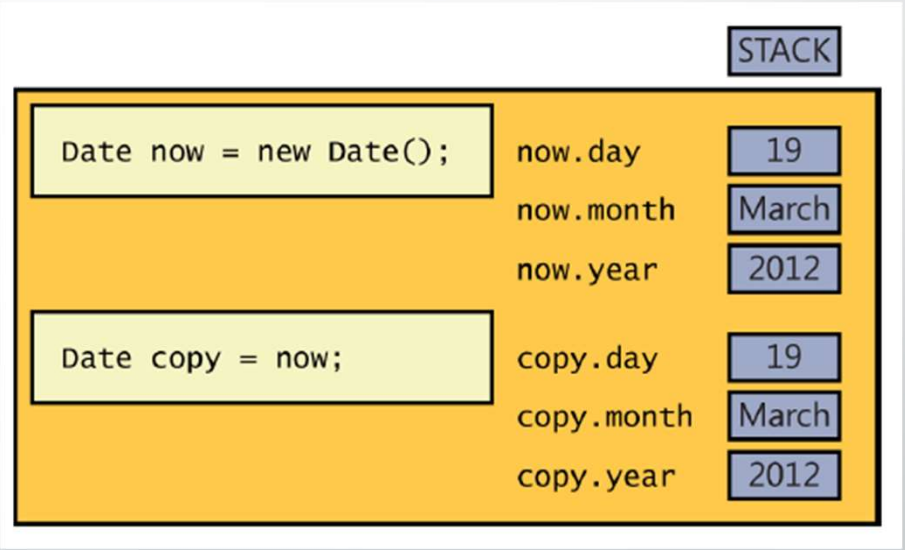
Estrutura

Date now = new Date();

Date copy = now;

Date now;

Date copy = now; // compile-time error: now has not been assigned



Introdução ao C#

Parte II – *Arrays*

Arrays ou vetores

- Declarando um *array:*

int[] pins; // Personal Identification Numbers

- Um a*rray* não está limitado a tipo valores, podendo ser classes e enumerações:

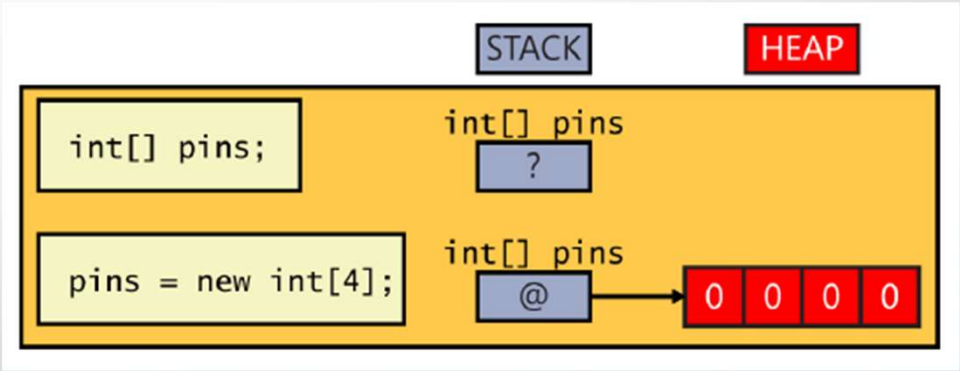
Date[] dates;

Arrays

- Instanciando um *array:*

pins = new int[4];

int[] pins = new int[4]{ 9, 3, 7, 2 };



Arrays

- Inicializações:

```
int[] pins = new int[3]{ 9, 3, 7, 2 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7 }; // compile-time error
int[] pins = new int[4]{ 9, 3, 7, 2 }; // OK
int[] pins = { 9, 3, 7, 2 };

Time[] schedule = { new Time(12,30), new Time(5,30) };
```

Acessando *arrays*

```
int myPin;
myPin = pins[2];

myPin = 1645;
pins[2] = myPin;
```

```
try
{
        int[] pins = { 9, 3, 7, 2 };
        // error, the 4th and last element is at index 3
        Console.WriteLine(pins[4]);}
catch (IndexOutOfRangeException ex)
{
        …
}
```

Iterando um *array*

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```

Iterando um *array*

- Uma outra forma de iterar um *array* é utilizando a palavra chave *foreach*:

```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

foreach

- Um *foreach* itera todo o vetor, sem pular nenhum item ou fazer uma iteração parcial.
- Um *foreach* sempre vai do índice 0 até o (comprimento – 1)
- Se for preciso saber o índice do objeto no *loop*, use *for*
- Se for preciso modificar o objeto no *loop*, use *for*

Passando *arrays* como parâmetro

```
public void ProcessData(int[] data)
{
    foreach (int i in data)
    {
        …
    }
}
```

Array como retorno de função

```
public int[] ReadData()
{
        Console.WriteLine("How many elements?");
        string reply = Console.ReadLine();
        int numElements = int.Parse(reply);
        int[] data = new int[numElements];
        for (int i = 0; i < numElements; i++)
        {
            Console.WriteLine("Enter data for element {0}", i);
            reply = Console.ReadLine();
            int elementData = int.Parse(reply);
            data[i] = elementData;
        }
        return data;
}
```

- E no código chama-se:

```
int[] data = ReadData();
```

Copiando *arrays*

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias and pins refer to the same array instance
```

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
        copy[i] = pins[i];
}
```

Copiando *arrays*

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

Arrays multidimensionais

- Pode-se criar *arrays* com mais de uma dimensão (matrizes, por exemplo):

```
int[,] items = new int[4, 6];

items[2, 3] = 99;            // set the element at cell(2,3) to 99
items[2, 4] = items [2,3]; // copy the element in cell(2, 3) to cell(2, 4)
items[2, 4]++;               // increment the integer value at cell(2, 4)
```

Arrays multidimensionais

- Podem ter mais de duas dimensões:

```
int[, ,] cube = new int[5, 5, 5];
cube[1, 2, 1] = 101;
cube[1, 2, 2] = cube[1, 2, 1] * 3;
```

Arrays irregulares (*jagged*)

```
int[,] items = new int[4, 40]; //Regular array

int[][] items = new int[4][];
int[] columnForRowo = new int[3];
int[] columnForRow1 = new int[10];
int[] columnForRow2 = new int[40];
int[] columnForRow3 = new int[25];
items[0] = columnForRowo;
items[1] = columnForRow1;
items[2] = columnForRow2;
items[3] = columnForRow3;
```

Introdução ao C#

Parte II – Herança

Herança

- É uma relação entre classes.
- Permite estabelecer relação de herança (como por exemplo entre cavalos e mamíferos) e também padrões comuns ao tipo mais alto.
- Exemplo: todos mamíferos mamam, porém baleias nadam, ao passo que um cavalos trotam.

Herança

- A sintaxe é bem simples para estabelecer a herança entre duas classes:

class DerivedClass : BaseClass {...}

- É possível herdar apenas de uma classe, porém se a classe não for marcada como *sealed* é possível derivar uma classe de uma que já foi derivada:

class DerivedSubClass : DerivedClass { ... }

Herança

```
class Mammal
{
        public void Breathe()
        {…}
        public void SuckleYoung()
        {…}
        …
}
```

```
class Horse : Mammal
{…
        public void Trot()
        {…}
}
```

```
class Whale : Mammal
{…
        public void Swim()
        {…}
}
```

Herança

- As classes anteriores podem ser usadas da seguinte forma:

```
Horse myHorse = new Horse();
myHorse.Trot();
myHorse.Breathe();
myHorse.SuckeYoung();
```

Herança

- Pode-se chamar o construtor da classe mãe na classe filha usando a palavra chave *base*:

```
class Mammal // base class
{...
        public Mammal(string name) // constructor for base class
        {...}
}
```

```
class Horse : Mammal // derived class
{...
        public Horse(string name)
                : base(name) // calls Mammal(name)
        {...}
}
```

Herança

- Se nada for especificado na classe filha, o compilador implicitamente chama o construtor padrão (vazio) da classe mãe.
- Isso funciona para classes com construtor público.

Herança

class Mammal { ...}

class Horse : Mammal{ ...}

class Whale : Mammal{ ...}

...

Horse myHorse = new Horse(...);

Whale myWhale = myHorse; // error - different types

Herança

- Porém, o seguinte código é permitido:

Horse myHorse = new Horse(…);
Mammal myMammal = myHorse; // legal, Mammal is the base class of Horse

- Porém o inverso não é:

Mammal myMammal = newMammal(…);
Horse myHorse = myMammal; // error

Herança

```
Horse myHorse = new Horse(...);
Mammal myMammal = myHorse;
myMammal.Breathe();      // OK - Breathe is part of the Mammal class
myMammal.Trot();         // error - Trot is not part of the Mammal class
```

Herança

- Se se desejar criar métodos novos na classe filha, como proceder?

```
class Mammal
{...
        public void Talk() // assume that all mammals can talk
        {...}
}
class Horse : Mammal
{...
        public void Talk() // horses talk in a different way from other mammals!
        {...}
}
```

Herança

- O código anterior não compila, porque existem duas funções com mesmo nome e quando houver a chamada da função, o compilador não sabe indicar qual função deve ser chamada.

- Para resolver esse problema, utiliza-se a palavra chave *new*.

Palavra chave *new*

```
class Mammal
{
        ...
        public void Talk()
        {...}
}
class Horse : Mammal
{
        ...
        new public void Talk()
        {...}
}
```

Métodos virtuais

- Métodos podem ser marcados com a palavra chave *virtual* para indicar que ele tem uma implementação mais simples para ser sobreposto (*overriding)* na classe filha. Exemplo:

```
namespace System
{ …
        class Object
        { …
                public virtual string ToString()
                {…}
        }
}
```

Palavra chave *override*

- Assim, pode-se sobrescrever o método *ToString* como:

```
class Horse : Mammal
{...
    public override string ToString()
    {...}
}
```

- Também é possível chamar o método da classe mãe:

```
public override string ToString()
{
    base.ToString();
    ...
}
```

Palavra chave *protected*

- Relembrando a questão do acesso, a palavra chave *protected* é utilizada no contexto de herança.
- Ela permite o acesso de uma variável pela classe e suas filhas apenas.

Introdução ao C#

Parte II – Interfaces e Classes Abstratas

Interfaces

- Interfaces funcionam como classes.
- Elas permitem a descrição de um comportamento ou funcionamento.
- Não podem ser instanciadas, mas podem ser implementadas em outras classes. Diferentemente da herança, uma classe pode implementar várias interfaces.

Interfaces

- Por exemplo, suponha que deseja-se criar um *array* mas que retorne os elementos em ordem. É possível criar uma classe que armazene o *array e* *implementar* a interface adequada para isso.

Interfaces

- Um método para comparar os objetos seria o a seguir:

```
int CompareTo(object obj)
{
    // return 0 if this instance is equal to obj
    // return < 0 if this instance is less than obj
    // return > 0 if this instance is greater than obj
    ...
}
```

Interfaces

- Declara-se uma interface utilizando a palavra chave *interface*:

```
interface IComparable
{
    int CompareTo(object obj);
}
```

- Importante! Interfaces não contém campos nem dados.

Interfaces

```
interface ILandBound
{
        int NumberOfLegs();
}

class Horse : ILandBound
{...
        public int NumberOfLegs()
        {
                return 4;
        }
}
```

```
interface ILandBound
{
        …
}
class Mammal
{
        …
}
class Horse : Mammal , ILandBound
{
        …
}
```

Interfaces

- Usos:

```
Horse myHorse = new Horse(…);
ILandBound iMyHorse = myHorse; // legal

int FindLandSpeed(ILandBound landBoundMammal)
{…}

if (myHorse is ILandBound)
{
        ILandBound iLandBoundAnimal = myHorse;
}
```

Múltiplas interfaces

- Como dito, uma classe pode implementar várias interfaces:

class Horse : Mammal, ILandBound, IGrazable

{…}

Explicitamente implementando interfaces

- Suponha que tenha-se o seguinte código:

```
interface ILandBound
{
        int NumberOfLegs();
}
interface Ijourney
{
        int NumberOfLegs();
}
```

```
class Horse : ILandBound, Ijourney
{ ...
        public int NumberOfLegs()
        {
                return 4;
        }
}
```

- Qual interface está implementada?

Explicitamente implementando interfaces

- Para tal, precisa-se implementar as interfaces explicitamente:

```
class Horse : ILandBound, IJourney
{...
        int ILandBound.NumberOfLegs()
        {
                return 4;
        }
        int IJourney.NumberOfLegs()
        {
                return 3;
        }
}
```

Explicitamente implementando interfaces

- Para obter o valor, precisa-se explicitamente converter o objeto para a interface desejada;

```
Horse horse = new Horse();
...
IJourney journeyHorse = horse;
int legsInJourney = journeyHorse.NumberOfLegs();
ILandBound landBoundHorse = horse;
int legsOnHorse = landBoundHorse.NumberOfLegs();
```

Classes abstratas

- Considere o seguinte exemplo:

```
class Horse : Mammal, ILandBound, IGrazable
{...
        void IGrazable.ChewGrass()
        {
                Console.WriteLine("Chewing grass");
                // code for chewing grass
        };
}
```

```
class Sheep : Mammal, ILandBound, IGrazable
{
        ...
        void IGrazable.ChewGrass()
        {
                Console.WriteLine("Chewing grass");
                // same code as horse for chewing grass
        };
}
```

- Código repetido! Não é uma boa maneira de programar!

Classes abstratas

- Pode-se resolver o problema com uma classe:

```
class GrazingMammal : Mammal, IGrazable
{...
        void IGrazable.ChewGrass()
        {
                // common code for chewing grass
                Console.WriteLine("Chewing grass");
        }
}
class Horse : GrazingMammal, ILandBound {...}
class Sheep : GrazingMammal, ILandBound {...}
```

Classes abstratas

- É uma boa solução, porém é possível instanciar a classe *GrazingMammal*, que não é desejável.
- Assim, a solução ideal é uma classe abstrata:

abstract class GrazingMammal : Mammal, IGrazable{ ...}

GrazingMammal myGrazingMammal = new GrazingMammal(...); // illegal

Classes abstratas

- Classes abstratas podem ter métodos abstratos:

```
abstract class GrazingMammal : Mammal, IGrazable
{
    abstract void DigestGrass();
    ...
}
```

- Esses métodos precisam ser necessariamente implementados nas classes derivadas!

Palavra chave *sealed*

- Classes e métodos marcados como *sealed* não podem ser derivados.
- Isso significa que não é possível derivar uma classe selada.
- E também que métodos selados não podem ser sobrescritos ou escondidos.

Palavras chave de acesso

Keyword	Interface	Abstract class	Class	Sealed class	Structure
abstract	No	Yes	No	No	No
new	Yes[1]	Yes	Yes	Yes	No[2]
override	No	Yes	Yes	Yes	No[3]
private	No	Yes	Yes	Yes	Yes
protected	No	Yes	Yes	Yes	No[4]
public	No	Yes	Yes	Yes	Yes
sealed	No	Yes	Yes	Required	No
virtual	No	Yes	Yes	No	No

1 An interface can extend another interface and introduce a new method with the same signature.
2 Structures do not support inheritance, so they cannot hide methods.
3 Structures do not support inheritance, so they cannot override methods.
4 Structures do not support inheritance; a structure is implicitly sealed and cannot be derived from.