
PICC

Programação Orientada por Objectos em C

Fernando Miguel Carvalho

Secção de Programação

Agenda

- OO em C
- Do C para C#
- Conceitos
- Herança e Polimorfismo
- UML

Demo01

Company_t

Struct

Fields

- first : Employee*
- nrOfEmployees : uint16

Employee_t

Struct

Fields

- category : EmployeeKind
- lunchExpense : __unnamed_000c_1
- name : char*
- next : Employee_t*
- nr : uint16
- salary : uint16

EmployeeKind_t

Enum

CONS
MAN

Expense_t

Struct

Fields

- month : uint8
- next : Expense_t*
- value : uint32
- year : uint16

Demo01...

- Novas categorias de EmployeeKind obrigam a modificação de funções e introdução de if/else → Pouca flexibilidade.
- Exemplo:

```
uint32 Employee_totalExpense(Employee * emp, uint8 month, uint16 year){
    Expense * exp;
    int total = emp->salary;
    if(emp->category == CONS){
        total += emp->lunchExpense.vouchers;
    }else{
        for(exp = emp->lunchExpense.invoices; exp != NULL; exp = exp->next){
            if(exp->month == month && exp->year == year)
                total += exp->value;
        }
    }
    return total;
}
```

Demo02

Company_t

Struct

Fields

- first : Employee*
- nrOfEmployees : uint16

Employee_t

Struct

Fields

- category : EmployeeKind
- lunchExpense : __unnamed_0017...
- methods : EmployeeMethods*
- name : char*
- next : Employee_t*
- nr : uint16
- salary : uint16

EmployeeKind_t

Enum

CONS
MAN

Expense_t

Struct

EmployeeMethods_t

Struct

Fields

- addExpense : void (*)(Employee *, int, int, int)
- free : void (*)(Employee *)
- toString : char *(*)(Employee *)
- totalExpense : unsigned int (*)(Employee *, uint8, uint16)

Demo02...

- Cada tipo de objecto define o comportamento das suas funções.
- Em tempo de compilação não é conhecida a função chamada.

```
Employee * Manager_new(uint16 nr, char * name){
    /*
    // The vtable is shared by all manager objects.
    // Overrides toString, totalExpense and addExpense.
    */
    static EmployeeMethods Manager_vtable = {
        Manager_free,
        Employee_toString, // Manager_toString,
        Manager_totalExpense,
        Manager_addExpense
    };
    /*
    // Instantiates a new object Employee
    */
    Employee * emp = Employee_new(nr, MAN, name);
    emp->methods = &Manager_vtable;
    emp->lunchExpense.invoices = NULL;
    emp->salary = 1500;
    return emp;
}
```

```
uint32 Company_totalExpense(Company * comp, ...){
    uint32 total = 0;
    Employee * emp;
    for(emp = comp->first; emp != NULL; emp = emp->next){
        total += emp->methods->totalExpense(emp, month, year);
    }
    return total;
}
```


Os métodos de Employee apresentam diferentes comportamentos (formas) consoante o objecto apontado – **Polimorfismo**.

Agenda


- OO em C
- Do C para C#
- Conceitos
- Herança e Polimorfismo
- UML

Main

- Em C# todas as funções são definidas no contexto de uma classe.



```
int main(void) {  
    ...  
}
```



```
class Program {  
    static void Main() {  
        ...  
    }  
}
```

Por convenção os nomes das funções
Começam por letra maiúscula.

Employee de C para C#

C

```
typedef struct Employee_t{
    EmployeeMethods * methods;
    struct Employee_t * next;
    uint16 nr;
    EmployeeKind category;
    char * name;
    uint16 salary;
    union{
        uint32 vouchers;
        Expense * invoices;
    } lunchExpense;
} Employee;
```

O nome da classe define um novo tipo.

C#

```
class Employee {
    public Employee next;
    protected UInt16 nr;
    protected EmployeeKind category;
    protected String name;
    protected Int16 salary;
    ...
}
```

Os campos têm acessibilidade.

Apenas tem acesso aos campos públicos de Employee.

```
class Program {
    static void Main() {
        Employee emp = ...;
        ✓ emp = emp.next;
        X UInt16 nrEmp = emp.nr;
    }
}
```

Employee... tipo referência

- Num contexto **safe** não é permitida a utilização de ponteiros em C#.
- Uma classe define um *reference type*.
 - Qualquer variável de um *reference type* guarda uma referência para o seu objecto;
 - Em C# um objecto de um *reference type* é alojado em memória dinâmica;
 - Um objecto de um *reference type* é instanciado com o operador **new**;
 - ‘.’ é o operador de acesso a **membros** (**campos** ou **métodos**).

Employee... layout...

C#

```
class Employee {  
    public Employee next;  
    protected UInt16 nr;  
    protected EmployeeKind category;  
    protected String name;  
    protected Int16 salary;  
    ...  
}
```

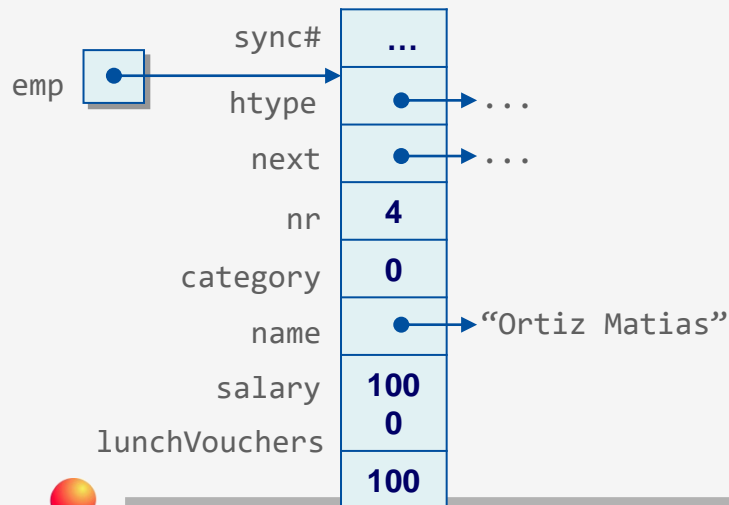
A classe Consultant estende Employee

C#

```
class Consultant: Employee {  
    protected UInt16 lunchVouchers;  
}
```

```
Employee emp = new Consultant(4, "Ortiz Matias");
```

C#



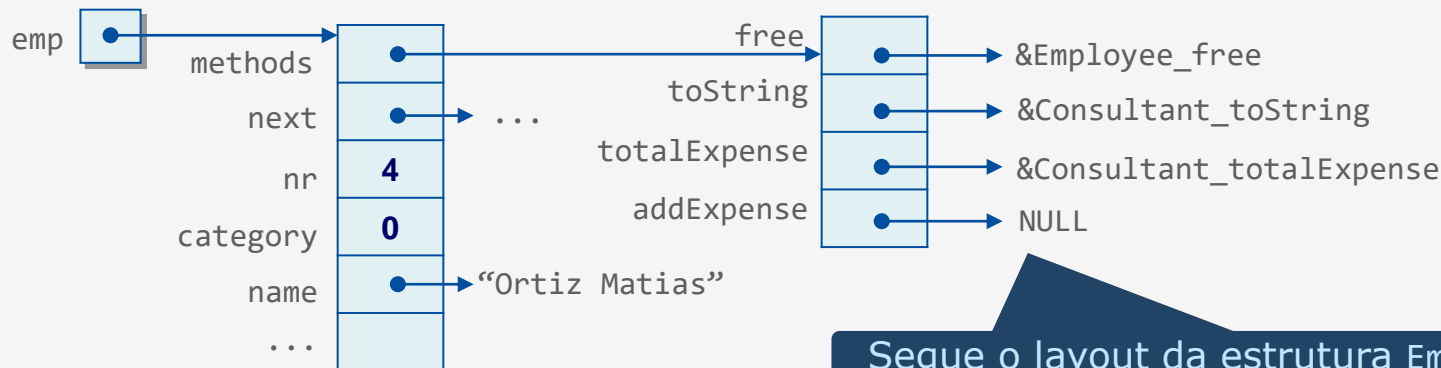
Um objecto da classe Consultant tem todos os campos definidos por Employee e Consultant.

Employee... métodos em C

```
typedef struct Employee_t{  
    EmployeeMethods * methods;  
  
    ...  
} Employee;
```

```
typedef struct EmployeeMethods_t{  
    void (* free) (Employee *);  
    char * (* toString) (Employee *);  
    uint32 (* totalExpense) (Employee *, uint8 month, uint16 year);  
    void (* addExpense) (Employee *, int value, int month, int year);  
} EmployeeMethods;
```

```
Employee * emp = Consultant_new(4, "Ortiz Matias");  
printf(emp->methods->toString(emp));
```



Segue o layout da estrutura `EmployeeMethods_t`

Employee... métodos em C#

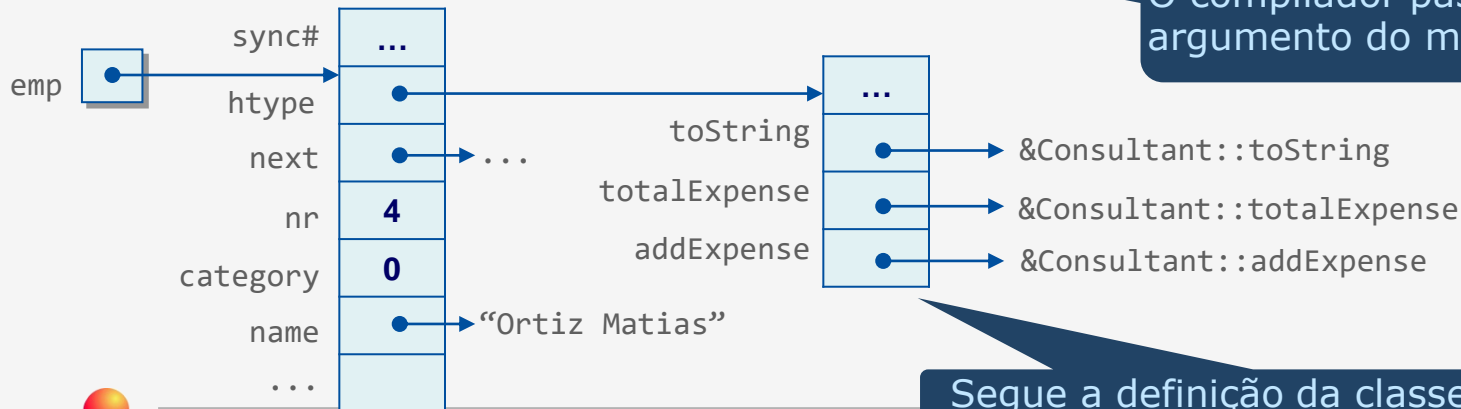
Todos os métodos de instância têm um 1º parâmetro implícito **this**.

```
class Employee {  
    ...  
    public Employee(UInt16 nr, ...) {...}  
    public String ToString() {  
        String type_desc = this.category.ToString();  
        String str = String.Format("{0} {1} {2} {3}",  
            type_desc, this.nr, this.name, this.salary);  
        return str;  
    }  
    public Int32 TotalExpense(Byte month, ...) {...}  
    public void AddExpense(Int32 value, ...) {...}  
}
```

```
Employee emp = new Consultant(4, "Ortiz Matias");  
Console.WriteLine(emp.ToString());
```

C#

O compilador passa '**emp**' como 1º argumento do método.



Segue a definição da classe `Consultant`

Employee... Construtor

```
class Employee {  
    ...  
    public Employee(UInt16 nr, EmployeeKind category, String name) {  
        this.nr = nr;  
        this.category = category;  
        this.name = name;  
    }  
}
```

```
Employee emp = new Employee(4, EmployeeKind.Consultant, "Ortiz Matias");  
Console.WriteLine(emp.ToString());
```

- Na **instanciação** de uma classe é invocado um **construtor** responsável pela inicialização dos campos do objecto criado.
- Uma classe que não define um construtor explicitamente, tem um construtor por omissão sem parâmetros.

Enumerado

```
typedef enum EmployeeKind_t{CONS = 0, MAN = 1} EmployeeKind;  
static char * EmployeeKindDesc [] = {"Consultor", "Manager"};
```



```
printf(EmployeeKindDesc[emp->category]);
```

Consultor

```
enum EmployeeKind {Consultant, Manager};
```



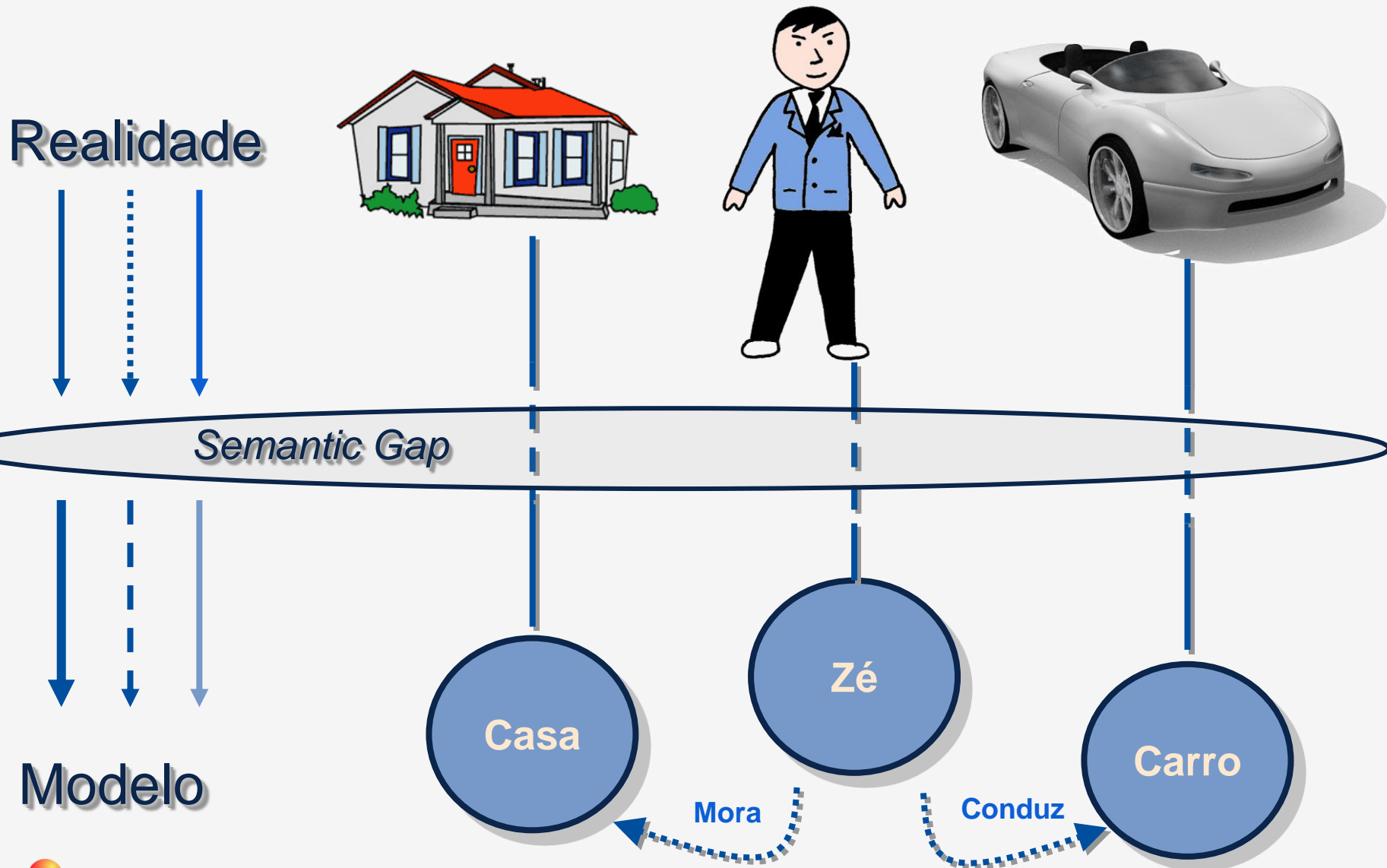
```
Console.WriteLine(emp.category);
```

Consultant

Agenda

- OO em C
- Caso prático do C para C#
- **Conceitos**
- Herança e Polimorfismo
- UML

Paradigma da Programação Orientada por Objectos



Paradigma da programação Orientada por Objectos

“... the biggest point of the object revolution [is] to reduce the semantic gap between the models we use in computer programming languages (...) and the conceptual models we use in thinking about the world normally.”

James Martin, Principles of Object Oriented

- Os conceitos base de POO são:
 - Objecto (instância)
 - Classe (modelo)
 - Herança
 - Polimorfismo
- O sucesso de uma modelação OO reside na capacidade de perceber o mundo real para se conseguir modelar esse mundo, a uma determinada escala...
- Ou seja, só é possível efectuar uma modelação coerente, quando a realidade é compreendida

Paradigma da Programação Orientada por Objectos



Abstract Data Types

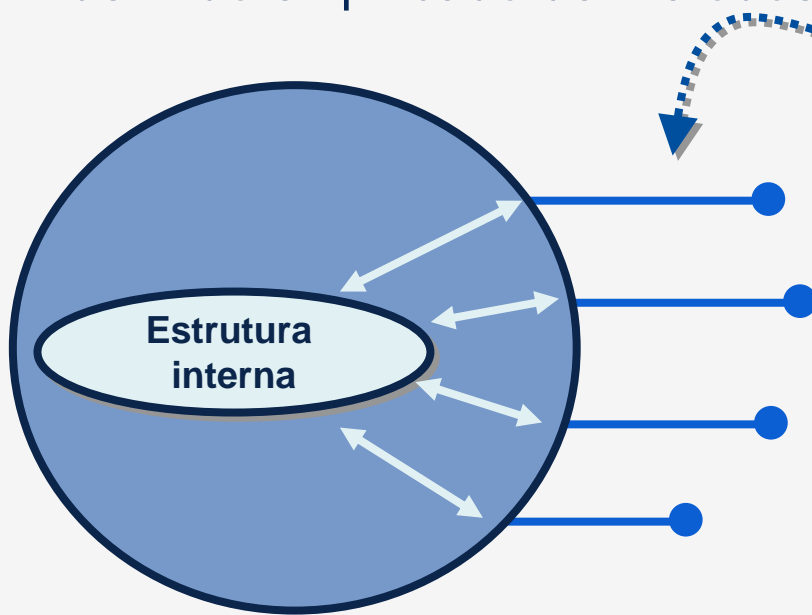
- Um *abstract data type* (ADT) é uma extensão do conceito de *user defined type* (UDT).
- O C tem construções que permitem a definição de novos tipos:
 - e.g. `enum`, `struct`
- Mas estas construções não permitem ter ADT



O ADT acrescenta à capacidade de definição de novos tipos o conceito de encapsulamento

Abstract Data Types

- O encapsulamento permite esconder os pormenores de implementação.
- Todas as acções são disponibilizadas através de um conjunto bem definido e tipificado de métodos – a interface.



A interacção é sempre feita **apenas** através da interface pública

Abstract Data Types e Objectos

- Em C# a definição de ADT's pode ser feita através de classes (*class*), que resulta na definição de um novo tipo.
- Uma classe representa um conjunto de características (*campos*) e comportamento (*métodos*) comuns a todos os objectos desse tipo.
- Ou seja, é um esquema que define o que é comum a todos os objectos de um determinado tipo.
- O encapsulamento é suportado pelos especificadores de **acessibilidade**:
 - *public*;
 - *private*;
 - *protected*.

Classe vs objecto

- Um objecto é uma entidade em tempo de execução que encapsula **dados** (campos) e **operações** (métodos) sobre esses dados.
- Uma classe é uma entidade em tempo de compilação que define um conjunto de especificações (campos e métodos) para um tipo abstracto de dados.
- Em tempo de execução são, normalmente, evocados os métodos que consultam e alteram o estados dos objectos.
- Todos os objectos têm uma identidade única – dada pela zona de memória onde residem.
- Os objectos são uma **instância** de uma dada classe que define o seu tipo.

Suporte OOP no C#

- **ADT:** Dá suporte ao conceito de classe/objecto

Keywords: `class`

- **Herança:** A capacidade de definir relações *is-a*, mas não só...
- **Polimorfismo:** A capacidade de, através de referências de tipos de classes mais abstractas, representar o comportamento das classes concretas que referenciam. Assim, um mesmo método pode apresentar várias formas, de acordo com seu contexto

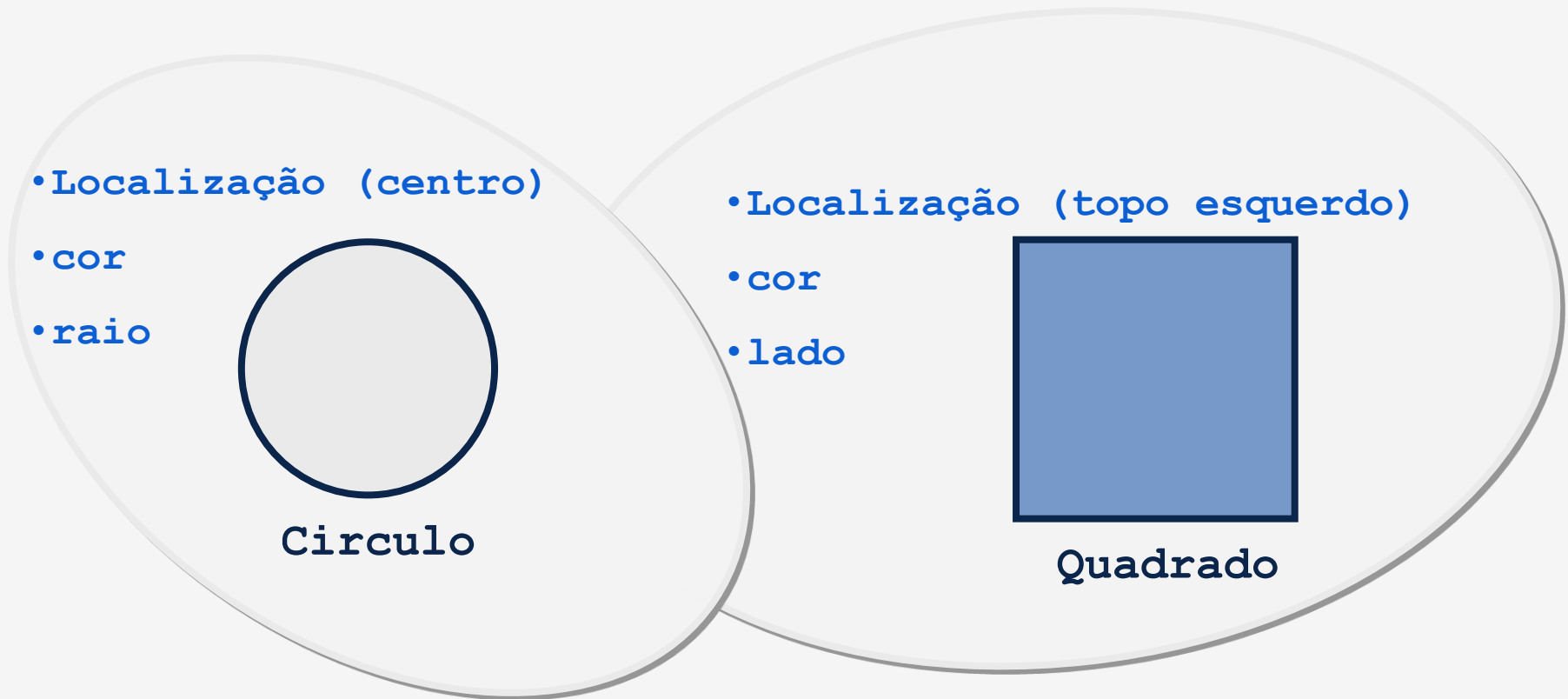
Keywords: `virtual`

Agenda

- OO em C
- Caso prático do C para C#
- Conceitos
- Herança e Polimorfismo
- UML

Herança

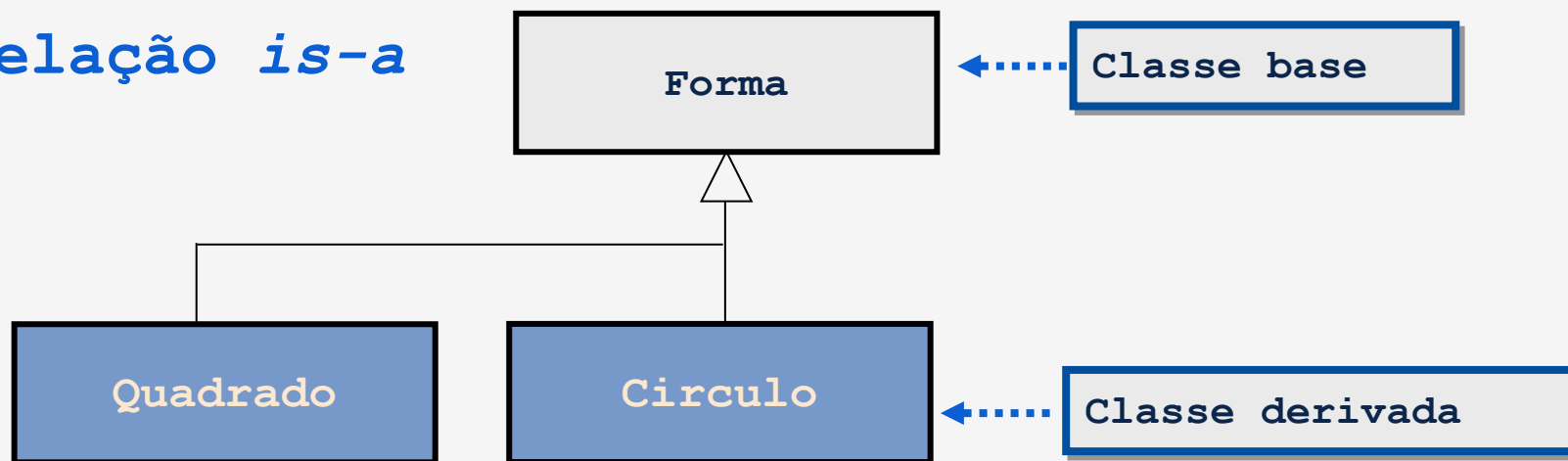
- Consideremos a modelação dos seguintes conceitos:



- Verifica-se que existe parte dos campos que é comum;
- Ambos são uma especialização de um conceito mais lato - *Forma*.

Herança

Relação *is-a*



- A herança é um mecanismo que permite a definir uma relação *is-a* entre classes, possibilitando que a classe derivada acrescente algo à classe base;
- Por isso, a classe derivada é mais específica e, normalmente, mais rica, quer em interface, quer em campos.

Herança – classe Forma

Se uma classe **não** é declarada explicitamente como **derivada** (herda) de outra classe, então esta deriva implicitamente de `System.Object`.

```
abstract class Forma
```

```
{  
    ...  
}
```

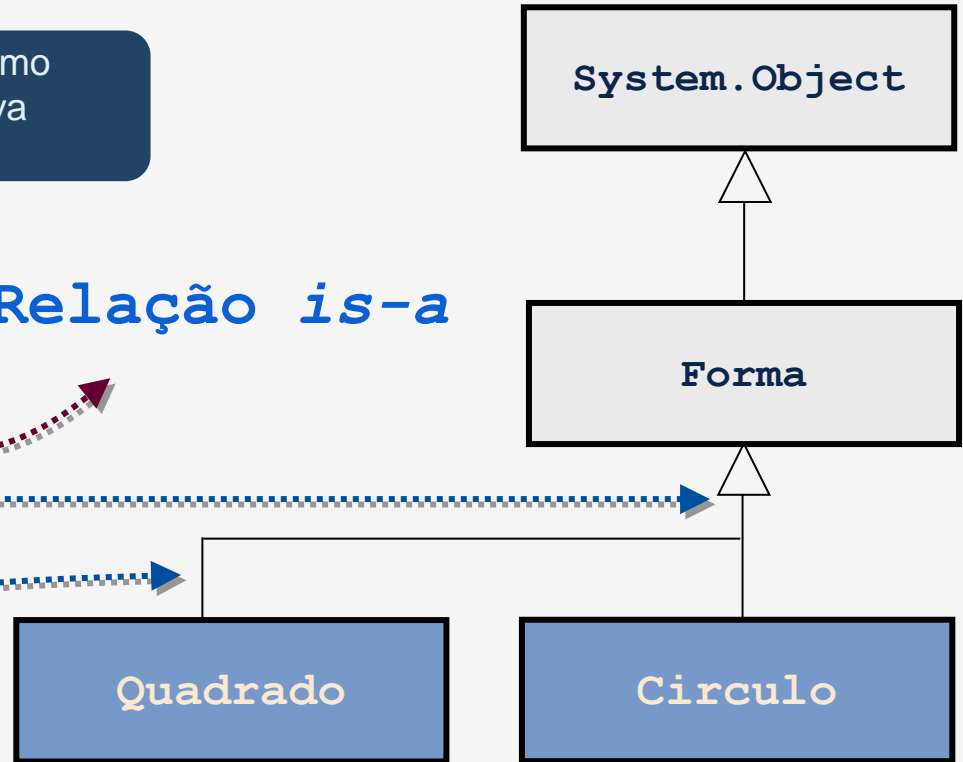
```
class Circulo: Forma
```

```
{  
    ...  
}
```

```
class Quadrado: Forma
```

```
{  
    ...  
}
```

Relação *is-a*

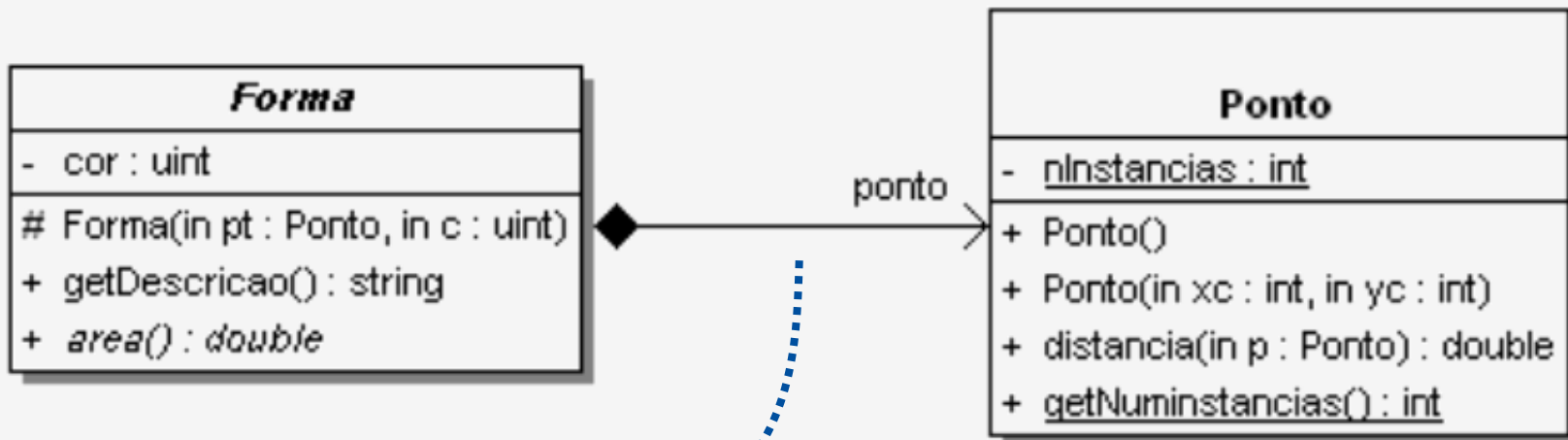


Classe System.Object

- Em .Net todos os tipos herdam, directa (e.g. Forma) ou indirectamente (Quadrado e Circulo) de System.Object.
→ Logo, qualquer objecto de qualquer classe é do tipo System.Object.
- Se uma classe **não** é declarada explicitamente como **derivada** (herda) de outra classe, então esta deriva implicitamente de System.Object.
- System.Object define os seguintes métodos de instância, herdados por todas as classes:

```
public virtual bool Equals(object) ;  
public virtual int GetHashCode() ;  
public virtual string ToString() ;  
protected virtual void Finalize() ;  
public Type GetType() ;
```

Associação



```
abstract class Forma{
    protected Ponto ponto;
    protected UInt32 cor;
    public Forma(Ponto pt, UInt32 c){
        this.ponto = pt;
        this.cor = c;
    }
}
```

```
class Ponto {
    private static int nInstancias;
    public int x,y;
    public Ponto(){
        ++nInstancias;
    }
    public Ponto(int xc,int yc):this(){
        this.x = xc;
        this.y = yc;
    }
    public double distancia(Ponto p){...}
    public static int getNumInstancias(){
        return Ponto.nInstancias;
    }
}
```

Chamada aos construtores da base

- Quando uma instância da classe derivada é criada, é necessário evocar o construtor da classe base sempre que:
 - Ela não apresente um construtor sem parâmetros, ou
 - É pretendido especificar qual o construtor a evocar.

```
public Quadrado(UInt32 l, Ponto p, UInt32 c) {...} //erro
```

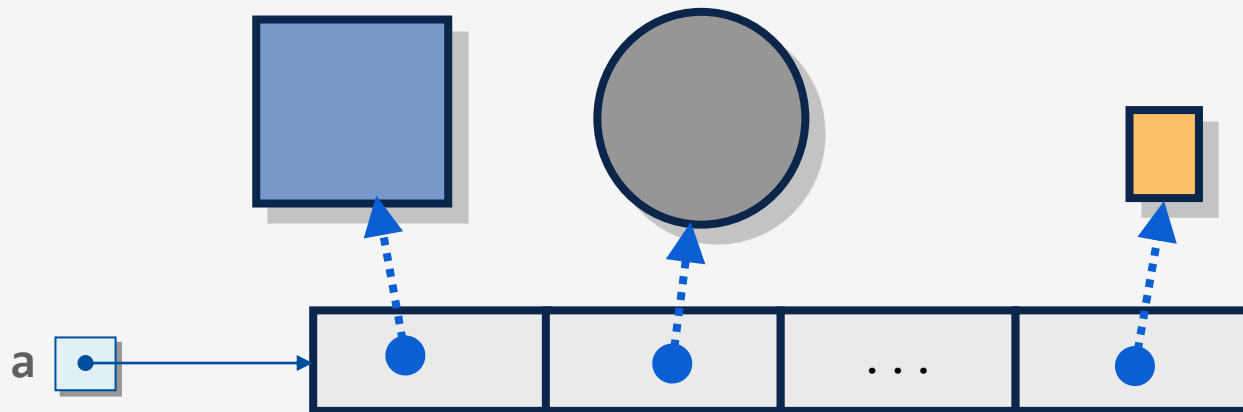
- O construtor de `Quadrado` não explicita a evocação do construtor de `Forma`, e `Forma` não tem construtor sem parâmetros:

```
public Forma(Ponto pt, UInt32 c) {  
    this.ponto = pt; this.cor = c;  
}
```

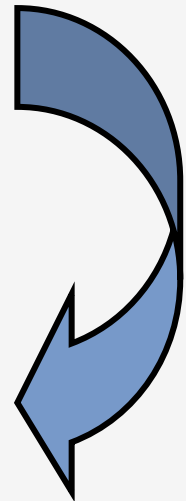
```
public Quadrado(UInt32 l, Ponto p, UInt32 c):base(p, c){  
    this.lado = l;  
}
```

Contentor genérico

- Pretende-se ter um contentor genérico de formas (independente do seu tipo real)



```
Forma [] a = new Forma[MAX];  
Circulo c = new Circulo (10, Ponto(1,1), 1);  
Quadrado q = new Quadrado (10, Ponto(1,1), 1);  
a[0] = c;  
a[1] = q;  
...
```



Contentor genérico

- Embora contendo referências para `Forma`, através da herança, um `Circulo` e um `Quadrado` são ambos do tipo `Forma`.
- Assim, espera-se que o comportamento de cada um dos objectos armazenados seja coerente com o seu tipo.
- No entanto...

```
Forma [] a = new Forma[MAX];  
Circulo c = new Circulo (10, Ponto(1,1), 1);  
Quadrado q = new Quadrado (10, Ponto(1,1), 1);  
a[0] = c;  
a[1] = q;  
...  
Console.WriteLine(a[0].getDescricao());
```

Forma

Polimorfismo

- Para que o comportamento de alguns métodos sejam correspondente ao tipo real do objecto (e não ao tipo da referência utilizada) é necessário definir na classe base que o comportamento será polimórfico, ou seja:
 - `virtual string getDescricao();`
- Se na classe base um método é virtual, o mesmo acontece nas classes derivadas.

Polimorfismo...

- Quando numa classe derivada se define um método com uma assinatura igual a um método virtual da classe base, pode-se redefinir o comportamento do método – **override**.

```
class Forma{  
    ...  
    public virtual string getDescricao(){  
        ...  
    }  
}
```

```
class Circulo{  
    ...  
    public override string getDescricao(){  
        return "Circulo";  
    }  
}
```

```
Forma [] a = new Forma[MAX];  
Circulo c = new Circulo (10, Ponto(1,1), 1);  
Quadrado q = new Quadrado (10, Ponto(1,1), 1);  
a[0] = c;  
a[1] = q;  
...  
Console.WriteLine(a[0].getDescricao());
```

Circulo

demo

Polimorfismo

Métodos abstractos

- Pretende-se que todas as formas respondam ao método `area`.
- No entanto, o cálculo da área depende do tipo de forma – terá de ser polimórfico.
- Mas, dada a definição de `Forma`, qual o código que o método terá?

```
class Forma{
    protected Ponto ponto;
    protected UInt32 cor;
    public Forma(Ponto pt, UInt32 c){
        this.ponto = pt; this.cor = c;
    }
    public virtual double area() { ??? }
}
```

Métodos abstractos...

- Não faz sentido existir uma implementação do método na classe Forma.
- O método será abstracto, fazendo com que a classe `Forma` passe a ser abstracta, i.e., não pode ter instâncias:

```
abstract double area();
```

- As classes derivadas que não sejam também abstractas têm de fornecer uma implementação para esse método:

```
class Círculo:Forma{  
    public override double area(){ return raio*raio*M_PI;}  
    ...}
```

```
class Quadrado:Forma{  
    public override double area(){ return lado*lado;;}  
    ...}
```

Métodos abstractos...

```
Forma [] a = new Forma[MAX];  
Circulo c = new Circulo (10, Ponto(1,1), 1);  
Quadrado q = new Quadrado (10, Ponto(1,1), 1);  
a[0] = c;  
a[1] = q;  
...  
Console.WriteLine(a[0].area());  
Console.WriteLine(a[1].area());
```

```
Class Circulo{  
    double area(){ return raio*raio*Math.PI;}  
    ...  
}
```

```
Class Quadrado{  
    double area(){ return lado*lado;};  
    ...  
}
```

```
314  
100
```

Agenda

- OO em C
- Caso prático do C para C#
- Conceitos
- Herança e Polimorfismo
- UML

Unified Modeling Language (UML)

Quando sublinhados,
os membros são estáticos.
A itálico indicam que
são abstractos.

Ponto

- nInstancias : int

+ Ponto()

+ Ponto(in xc : int, in yc : int)

+ distancia(in p : Ponto) : double

+ getNumInstancias() : int

Nome

campos

Tipo dos parâmetros

Tipo de retorno

Métodos

Visibilidade:

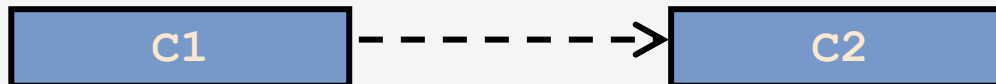
- Privada

Protegida

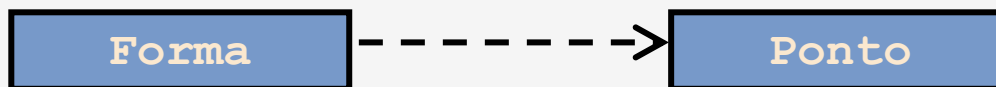
+ Pública

Unified Modeling Language (UML)... Dependência

- Relação entre entidades onde:
 - uma operação de uma entidade depende da presença de outra entidade;
 - alterações numa entidade podem afectar o comportamento da outra entidade.
- Uma situação usual de dependência é a relação “usar”:
 - A classe C1 depende da classe C2, se C1 usar C2 em lugares como parâmetros, variáveis locais ou tipo de retorno de métodos.



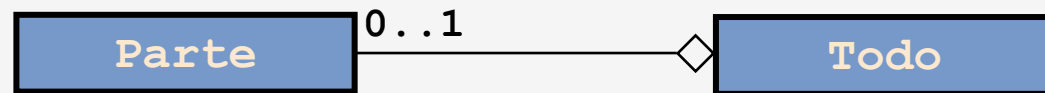
- Exemplo: o construtor da classe Forma recebe um ponto como parâmetro logo depende da classe Ponto.



Unified Modeling Language (UML)... Associação

- Uma **associação** é um caso particular de **dependência**.
- A relação de **associação** pode ser de **agregação**, ou **composição**.

- Agregação (◇):



- Composição (◆) :



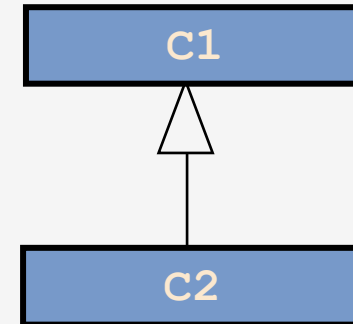
Cardinalidade. Quantas instâncias de **Parte** são contidas em **Todo**:

1	Exactamente uma
0..1	Zero ou uma
1..*	Uma ou várias
0..*	Zero ou várias

Unified Modeling Language (UML)... Herança

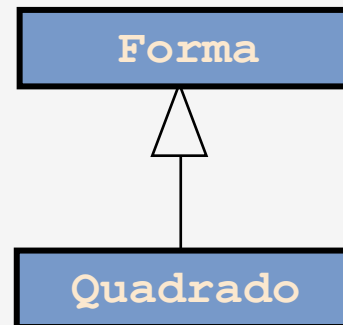
- Quando uma classe **C2 estende a classe C1**, a classe C2 é conhecida como **subclasse** de C1 e a classe C1 é conhecida como a **super classe** (ou **classe base**) de C2.

Herança

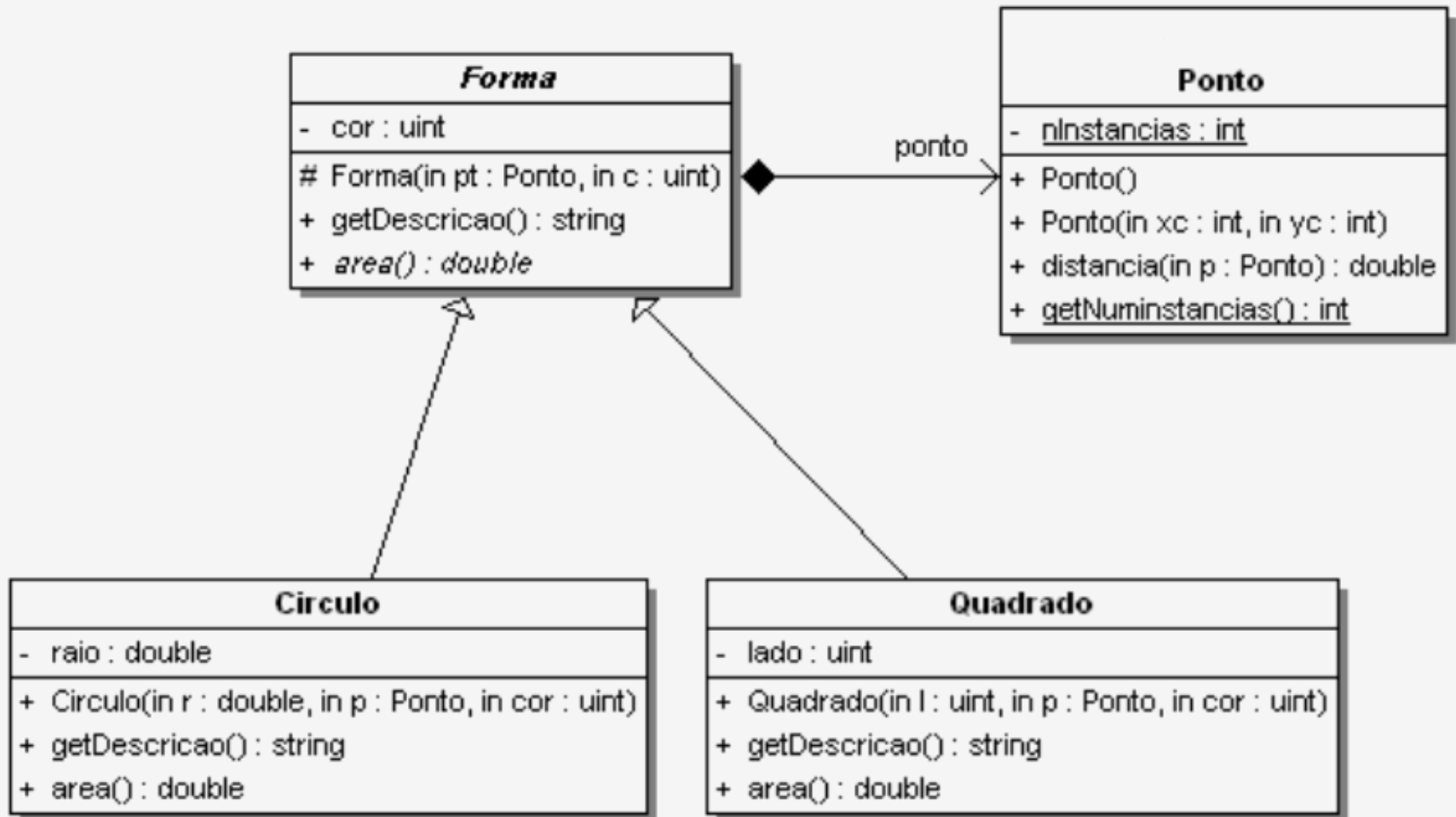


- Exemplo:

```
class Quadrado:Forma{
...
}
```



Hierarquia Forma



Referências

- “Object-Oriented Software Development Using Java”,
Xiaoping Jia, Addison-Wesley
- “Fundamentals of Object-Oriented Design in UML”,
Meilir Page-Jones, Larry L. Constantine, Addison-Wesley
- Jeffrey Richter, “CLR via C#, Second Edition”,
Microsoft Press; 2nd edition, 2006
- Don Box, “Essential .NET, Volume I:
The Common Language Runtime”,
Addison-Wesley Professional; 1st edition, 2002

