

# ORIENTAÇÃO À OBJETOS C++

Prof. Rafael de Santiago, MSc.

# ORIENTAÇÃO A OBJETOS COMO PARADIGMA



# CLASSES

```
class Aluno{  
    //...  
};
```

# CLASSES

Nome da classe.

Usem a notação Camel Case:

<http://pt.wikipedia.org/wiki/CamelCase>

```
class Aluno{  
    //...  
};
```

# CLASSES

```
class Aluno{  
    //...  
};
```

Neste espaço temos o escopo da classe.  
Tudo que pertence a ela deve ser descrito  
nesta região.

# ATRIBUTOS

```
#include <iostream>
```

```
using namespace std;
```

```
class Aluno{  
public:
```

```
    int codMatricula;
```

```
    string nome
```

```
};
```

Um atributo é uma característica relacionada às instâncias da classe.

Um Aluno tem diversas características, mas no nosso domínio, estas são suficientes.

# MÉTODOS

```
#include <iostream>

using namespace std;

class Aluno{
public:
    int codMatricula;
    string nome;

    void imprimir(){
        cout<<"Nome:"<<nome;
    }

};
```

Método geralmente é utilizado para especificar um comportamento dentro de uma classe que utiliza seu atributos ou não.

# MÉTODOS

```
#include <iostream>

using namespace std;

class Aluno{
public:
    int codMatricula;
    string nome;

    void imprimir(){
        cout<<"Nome:"<<nome;
    }

};
```

Todo método é possui um nome e uma assinatura. Verifique que o método abaixo possui o nome “imprimir” e assinatura “void imprimir()”



# MÉTODOS

```
class Aluno{  
public:  
    int codMatricula;  
    string nome;  
  
    void imprimir() {  
        cout<<"Nome:"<<nome;  
    }  
  
};
```

Note que o retorno deste método é void, ou seja, vazio, portanto não possui retorno.

Ao ser executado ele deverá imprimir na tela o valor do atributo nome.

# MÉTODOS

Note também que, ao acessar o atributo nome, pode-se utilizar o “this”.

Esta referência (this) é utilizada para acessar atributos e métodos da classe em qualquer escopo de método ou construtor pertencente a classe que está sendo codificada

```
class Aluno{  
public:  
    int codMatricula;  
    string nome;  
  
    void imprimir() {  
        cout<<"Nome:"<<this->nome;  
    }  
  
};
```

# MÉTODOS

```
class Aluno{
public:
    int codMatricula;
    String nome;
    void imprimir(){
        cout<<"Nome:"<<this->nome<<endl;
    }
    bool ehCodigo(int cod){
        if (cod == this->codMatricula){
            return true;
        }else{
            return false;
        }
    }
};
```

O retorno deste novo método é boolean, ou seja, esta função retorna ou verdadeiro ou falso.

# MÉTODOS

```
class Aluno{
public:
    int codMatricula;
    String nome;
    void imprimir() {
        cout<<"Nome:"<<this->nome;
    }
    bool ehCodigo(int cod) {
        if (cod == this->codMatricula) {
            return true;
        }else{
            return false;
        }
    }
};
```

Note que a função recebe um valor como parâmetro (chamado “cod”).

A função compara o “cod” com o atributo “codMatricula”. Se forem iguais retorna verdadeiro, caso contrário falso.

# PUBLIC, PRIVATE, PROTECTED

- O C++ possui alguns controles de acesso a membros de classes:
  - public
  - private
  - protected

# PUBLIC, PRIVATE, PROTECTED

- Níveis de acesso:

Modifier	Dentro da própria classe	Nas classes “filhas” (herança)	Outros contextos
public	Sim	Sim	Sim
protected	Sim	Sim	Não
private	Sim	Não	Não

# PUBLIC, PRIVATE, PROTECTED

Dentro de uma classe podemos acessar os próprios membros em qualquer escopo método ou construtor.

- Níveis de acesso:

Modifier	Dentro da própria classe	Nas classes “filhas” (herança)	Outros contextos
public	Sim	Sim	Sim
protected	Sim	Sim	Não
private	Sim	Não	Não

# PUBLIC, PRIVATE, PROTECTED

Dentro de uma subclasse, posso acessar todos os membros “public” e “protected” da classe pai

- Níveis de acesso:

Modifier	Dentro da própria classe	Nas classes “filhas” (herança)	Outros contextos
public	Sim	Sim	Sim
protected	Sim	Sim	Não
private	Sim	Não	Não



# PUBLIC, PRIVATE, PROTECTED

Em outros contextos, apenas pode-se acessar os membros públicos

- Níveis de acesso:

Modifier	Dentro da própria classe	Nas classes “filhas” (herança)	Outros contextos
public	Sim	Sim	Sim
protected	Sim	Sim	Não
private	Sim	Não	Não

# ESTADOS

```
class Aluno{  
public:  
    int codMatricula;  
    string nome;
```

```
    void alteraNome(string nome) {  
        this->nome = nome;  
    }
```

```
};
```

O estado é o valor do atributo. No método “alteraNome” o atributo “nome” altera seu valor de acordo com o especificado no parâmetro “nome”.

# ENCAPSULAMENTO

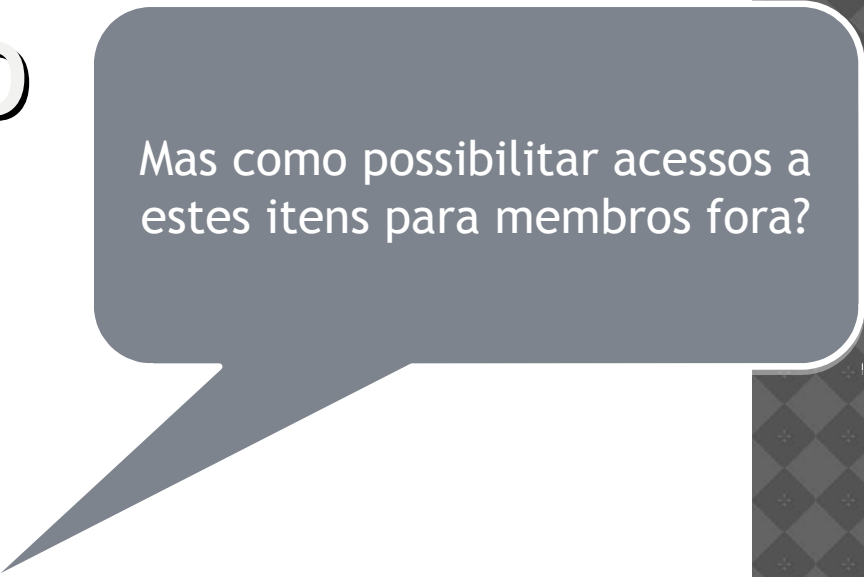
- Encapsulamento é uma boa prática para a programação Orientada à Objetos.
- Visa restringir os acessos aos atributos de uma classe para apenas os métodos da mesma.
- Para isto, emprega-se o método private, nos atributos.

# ENCAPSULAMENTO

```
class Aluno{  
private:  
    int codMatricula;  
    string nome;  
};
```

Os atributos passam a ter nível de acesso “private”

# ENCAPSULAMENTO



Mas como possibilitar acessos a estes itens para membros fora?

```
class Aluno{  
private:  
    int codMatricula;  
    string nome;  
  
};
```

# ENCAPSULAMENTO

Para possibilitar acessos aos atributos de uma classe, criamos os métodos “get” e “set”

```
class Aluno{  
private:  
    int codMatricula;  
    string nome;  
  
};
```

## ENCAPSULAMENTO

Os métodos “get” são utilizados para permitir acesso ao estado de um determinado atributo.

Utilizamos o nome “get” apenas por convenção. Poderíamos utilizar os nomes que quisermos.

```
class Aluno {  
private:  
    int codMatricula;  
    string nome;  
  
public:  
    int getCodMatricula() {  
        return this->codMatricula;  
    }  
};
```

# ENCAPS

```
class Alur
```

```
private:
```

```
    int cod
```

```
    string nome;
```

```
public:
```

```
    int getCodMatricula() {
```

```
        return this->codMatricula;
```

```
    }
```

```
    void setCodMatricula(int codigo) {
```

```
        this->codMatricula = codigo;
```

```
    }
```

```
};
```

Os métodos “set” são utilizados para alterar o estado de um determinado atributo.

Se não desejássemos troca do estado de um determinado atributo para membros fora desta classe, apenas não criamos este tipo de método

Utilizamos o nome “set” apenas por convenção. Poderíamos utilizar os nomes que quisermos.



# ENCAPSULA

Note que os métodos “gets” e “sets” possuem acessibilidade “public” para permitir que membros fora da classe possam alterar os atributos

```
class Aluno{  
private:  
    int codMatricula;  
    string nome;  
public:  
    int getCodMatricula() {  
        return this->codMatricula;  
    }  
  
    void setCodMatricula(int codigo) {  
        this->codMatricula = codigo;  
    }  
  
};
```

# OBJETOS

- Objetos são instâncias das classes.
- Ao comparar classes com objetos, podemos utilizar a analogia relativa a planejamento e construção de uma casa.
- Podemos dizer que a classe é como se fosse uma planta de uma casa. Ou seja, especifica como a casa deve ser
- Um objeto é uma casa em si. Várias casas podem ser construídas a partir de uma única planta. Vários objetos podem ser construídos a partir de uma única classe.

# OBJETOS

A partir da especificação da classe aluno, foram criados dois objetos. Um para gravar dados do aluno “Jorge Silva” e outro para o aluno “Luiz Silva”

```
#include <iostream>
using namespace std;
#include "aluno.h"

int main()
{
    Aluno j;
    j.setCodMatricula(137);
    j.setNome("Jorge Silva");
    Aluno p;
    p.setCodMatricula(139);
    p.setNome("Luiz Silva");
    return 0;
}
```

# CONSTRUTORES

- Construtores são muito semelhantes aos métodos, pois possuem escopo de instruções e podem ou não receber parâmetros, mas possuem particularidades:
  - não possuem retorno (inclusive não podem ser assinados com void, pois serão entendidos como métodos pelos compiladores/interpretadores)
  - são invocados no momento da instanciação (criação de um objeto)

# CONSTRUTORES

```
class Aluno{  
private:  
    int codMatricula;  
    string nome;  
  
};
```

Todas as classes possuem um construtor padrão.

Este construtor não possui parâmetros e não modifica a classe.

Apenas permite que objetos sejam criados, utilizando a especificação da classe.

# CONSTRUTORES

```
int main()  
{  
    Aluno j;  
    return 0;  
}
```

Observando a instância da classe (criação de um objeto) no “main”, podemos observar que, apesar de não especificarmos um construtor, há um implícito que poderá ser utilizado para criar objetos.

# CONSTRUTORES

- Vamos criar um construtor e tentar compreendê-lo na prática

# CONSTRUTORES

```
class Aluno{  
private:  
    int codMatricula;  
    string nome;  
public:  
    Aluno(int cod, string nome){  
        this->codMatricula = cod;  
        this->nome = nome;  
    }  
  
};
```

Note que a assinatura de um construtor possui como nome, o mesmo nome da classe. Não há retorno e possui especificação da visibilidade (na maioria das vezes, public). Podem ou não haver parâmetros que devem ser passados na criação de Objetos de uma classe.



# CONSTRUTORES

```
int main()  
{  
    Aluno j(137, "Jorge Silva");  
    return 0;  
}
```

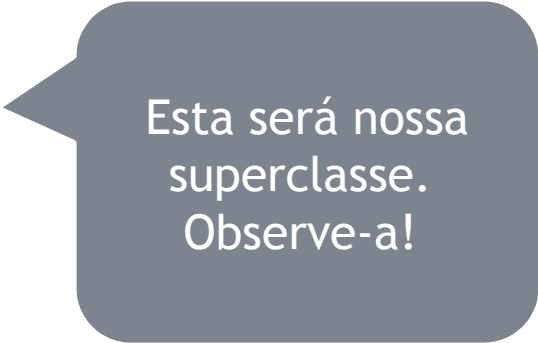
Observe que o construtor agora exige dois parâmetros, especificados previamente na classe.

# ASSOCIAÇÕES

# HERANÇA

- Quando duas ou mais classes que possuem atributos semelhantes e parece haver uma relação “familiar” (comum) entre as mesmas, utiliza-se o recurso de herança.
- Na herança possuímos dois níveis de classes as Superclasses (também chamadas de classes pai) e as Subclasses (também chamadas de classes filhos)

```
class Aluno{
private:
    int codMatricula;
    string nome;
    float mensalidade;
public:
    Aluno (float mensalidade){
        this->mensalidade = mensalidade;
    }
    int getCodMatricula() {
        return this->codMatricula;
    }
    void setCodMatricula(int codigo) {
        this->codMatricula = codigo;
    }
    float getMensalidade () {
        return this->mensalidade;
    }
};
```



Esta será nossa  
superclasse.  
Observe-a!

AlunoBolsista é subclasse de Aluno.  
Esta relação é explícita na  
assinatura da classe

```
class AlunoBolsista: public Aluno{
private
    float desconto;
public:
    AlunoBolsista (float m, float d):
        Aluno (m) {
        this.desconto = d;
    }
    float getMensalidadeDesconto () {
        return this->getMensalidade() -
            this->desconto;
    }
};
```

Note que o construtor é outro e recebe como parâmetro mensalidade e desconto.

```
class AlunoBolsista: pub Aluno{
private
    float desconto;
public:
    AlunoBolsista (float m, float d):
        Aluno(m) {
        this.desconto = d;
    }
    float getMensalidadeDesconto () {
        return this->getMensalidade() -
            this->desconto;
    }
};
```

Como o construtor de Aluno exige como parâmetro a mensalidade, deve-se utilizar a chamada do construtor de Aluno na assinatura da classe AlunoBolsista, passando a mensalidade por parâmetro

```
class AlunoBolsista: public Aluno{
private:
    float desconto;
public:
    AlunoBolsista (float m, float d):
        Aluno(m) {
        this.desconto = d;
    }
    float getMensalidadeDesconto () {
        return this->getMensalidade() -
            this->desconto;
    }
};
```

AlunoCota é subclasse de Aluno.

```
class AlunoCota: public Aluno{  
public:  
    AlunoCota (): Aluno (0.0) {  
    }  
};
```



Note que o construtor é outro e não recebe como parâmetro.

```
class AlunoCota: public Aluno{  
public:  
    AlunoCota (): Aluno (0.0) {  
    }  
};
```

Como o construtor de Aluno exige como parâmetro a mensalidade, deve-se utilizar o construtor de Aluno na assinatura da superclasse AlunoCota.

AlunoCota não possui mensalidade, por isso zero foi passado!

```
class AlunoCota: public Aluno {  
public:  
    AlunoCota (): Aluno (0.0) {  
    }  
};
```

Invocando construtores das classes Aluno, AlunoBolsista e AlunoCota. Deste modo temos 3 objetos. Um para um aluno em situação normal (Aluno), outro para bolsista (AlunoBolsista) e um para um aluno em regime especial (AlunoCota)

```
int main() {  
    Aluno n(2300.0);  
    AlunoBolsista j(790.0, 780.0);  
    AlunoCota s();  
  
    return 0;  
}
```

# DEPENDÊNCIA

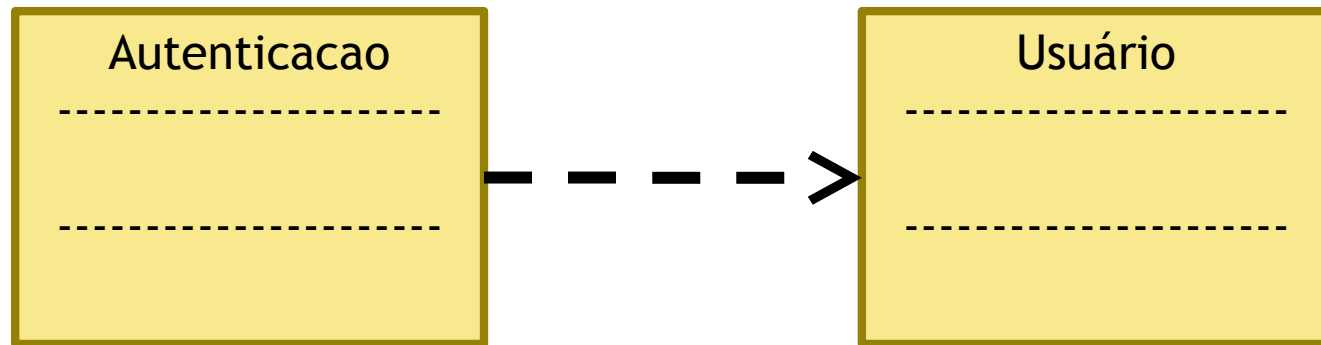
- Dependências são relações de uso
- Uma dependência indica que mudanças em um elemento (o “servidor”) podem afetar outro elemento (o “cliente”)
- Uma dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe

# DEPENDÊNCIA

- Dependências são relações de uso
- Uma dependência indica que mudanças em um elemento (o “servidor”) podem afetar outro elemento (o “cliente”)
- Uma dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe

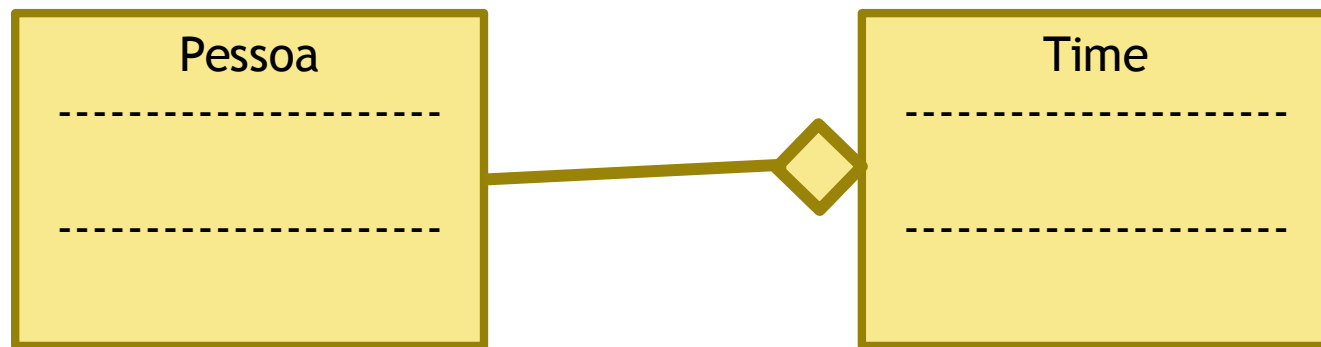
# DEPENDÊNCIA

Uma classe de autenticação usa a classe de usuário.



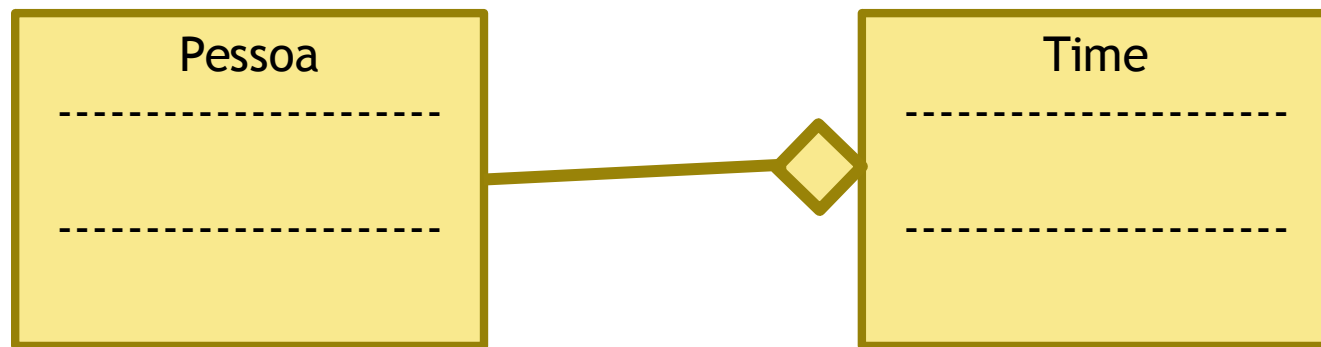
# AGREGAÇÃO

- Utilizada quando um item agrega outro, mas sem a necessidade de exclusividade total;



# AGREGAÇÃO

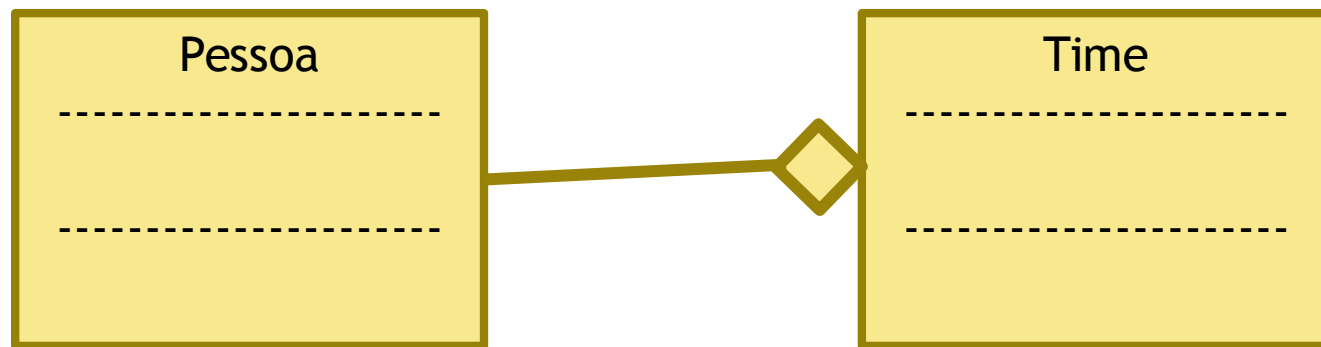
- Utilizada quando um item agrega outro, mas sem a necessidade de exclusividade total;





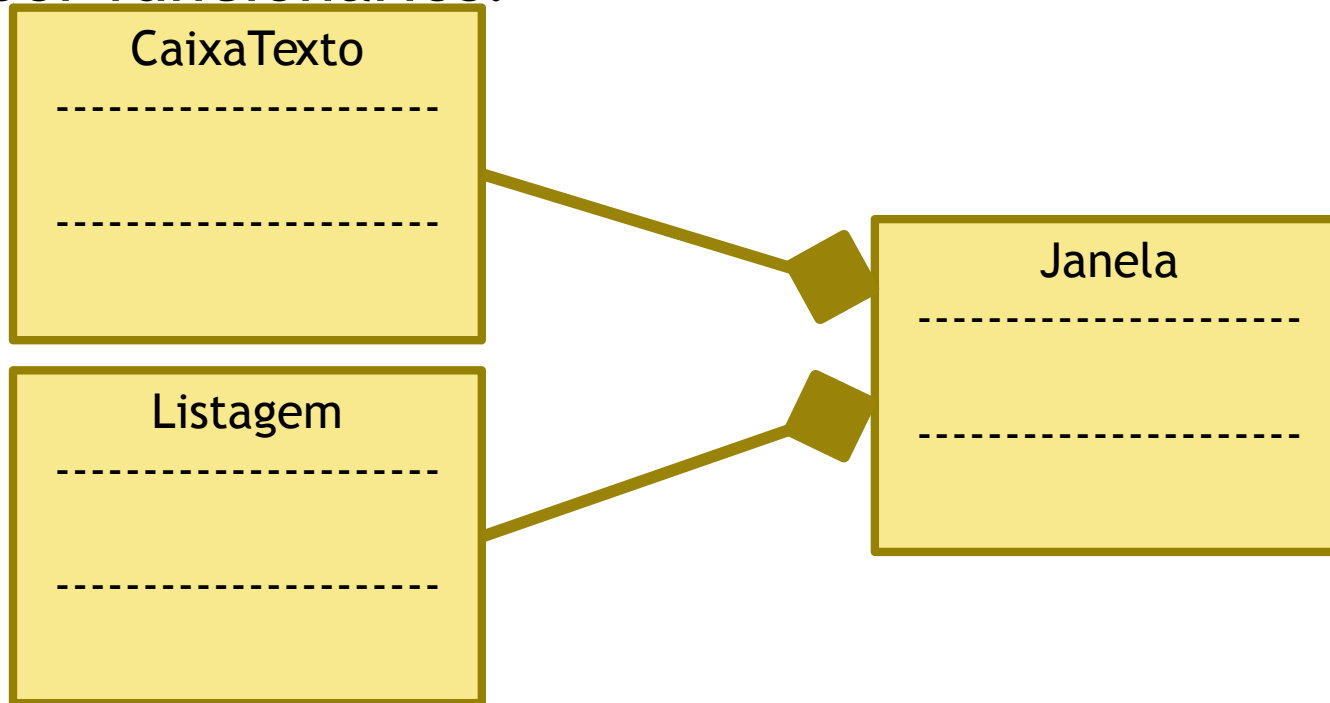
# AGREGAÇÃO

- Utilizada quando um item agrega outro, mas sem a necessidade de exclusividade total;



# COMPOSIÇÃO

- Idéia de compor algo. Por exemplo: uma Janela é composta por campos de texto, botões e menus. Uma empresa é composta por funcionários.



# COMPOSIÇÃO

- Idéia de compor algo. Por exemplo: uma Janela é composta por campos de texto, botões e menus. Uma empresa é composta por funcionários.

