INF 112: Programação II

Aula 19

- **Miscelânea (Operadores de bits e mais alguns conceitos de POO):**
  - **Operadores de bits**
  - **Atributos e métodos estáticos**
  - **Métodos *inline***
  - **Introdução a diagrama de classes**

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

Operadores de bits

Operadores de bits

- C++ tem seis operadores que atuam em bits individuais:

Operador	Nome	Descrição	Exemplo
&	E sobre bits	Os bits do resultado são 1 se os bits correspondentes nos dois operandos forem 1	0011 & 0101 → 0001
|	OU sobre bits	Os bits do resultado são 1 se pelo menos um dos bits correspondentes nos operandos for 1	0011 | 0101 → 0111
^	OU Exclusivo sobre bits	Os bits do resultado são 1 se exatamente um dos bits correspondentes nos dois operandos for 1	0011 ^ 0101 → 0110

Operadores de bits

Operador	Nome	Descrição	Exemplo
~	Comple-mento de bits	Os bits do resultado são 1 se os bits correspondentes no operando forem 0 e são 0 se forem 1	~0011 → 1100
<<	Shift de bits para esquerda	Desloca os bits do primeiro operando para a esquerda pelo número de bits especificado pelo segundo operando	01011 << 1 = 10110
>>	Shift de bits para direita	Desloca os bits do primeiro operando para a direita pelo número de bits especificado pelo segundo operando	01011 >> 2 = **010

➔ ** - o valor dos bits preenchidos à esquerda depende da máquina e se o valor é *signed* ou *unsigned*.

Operadores de bits

- Qual o resultado das operações abaixo?

a) `1 << 2`
b) `7 & 24`
c) `~20`

Operadores de bits

- Uma das aplicações comuns de operadores de bits é utilizar uma cadeia de bits contendo vários flags, que podem ser verificados ou configurados com “máscaras de bits”. Suponha por exemplo os seguintes valores que serão usados para acender indicadores luminosos no painel de um veículo:

```
#define MOTOR 1
#define COMBUSTIVEL 2
#define FAROL 4
#define OLEO 8
#define SETA_ESQUERDA 16
#define SETA_DIREITA 32
#define AR_CONDICIONADO 64
// Note que esses valores em binário são
// respectivamente 1, 10, 100, ..., 1000000.
```

Operadores de bits

- Suponha que `estado` é uma variável inteira que contenha o estado desses indicadores (ligado ou desligado). Assim, se seu valor é 20, estarão ligados os indicadores da seta esquerda e farol, pois `20 = 0010100`, ou seja, `4+16`.

- Um teste para saber se o farol está ligado seria:

```
if (estado & FAROL)  // note que o resultado
                     // independe do estado dos
                     // outros indicadores.
```

- Para ligar o indicador de combustível bastaria:

```
estado = estado | COMBUSTIVEL  // note que os
                               // demais continuam do
                               // modo como estavam.
```

Operadores de bits

- Para ligar mais de um indicador, podemos usar:

```
estado = estado | SETA_ESQUERDA | SETA_DIREITA
// Ativa pisca-alerta :)
```

- Esta sequência de ativação de bits usando o operador | é familiar para você?

- Lembre-se de que num exemplo de uma das aulas práticas sobre arquivos, fizemos:

```
fstream arq("arquivoaleat.bin",ios::in|ios::out|ios::binary);
```

- Portanto, aqueles flags de modo de abertura de arquivo seguem uma estrutura parecida com o exemplo do carro que vimos.

Atributos e métodos estáticos

Membros estáticos

➔ Uma classe pode conter *membros estáticos* (atributos e/ou métodos) ou, como também conhecidos, "membros de classe".

➔ O conteúdo de um membro estático é o mesmo para todos os objetos da classe e não é necessário instanciar um objeto para acessar o membro estático.

➔ Por exemplo, é possível manter um atributo estático de uma classe contendo um contador do número de objetos que foram instanciados:

```
class Teste {
  public:
    static int n;
    Teste () { n++; }
    ~Teste () { n--; }
};
...
```

Membros estáticos

```
...
int Teste::n=0; //temos que inicializar n globalmente

int main ()
{
    Teste a; // 1 instancia aqui.
    Teste b[5]; // + 5 instancias.
    Teste * c = new Teste; + 1 aqui.

    cout << a.n << endl; // 7 = 1+5+1.
    delete c; // Menos uma aqui (executa o destrutor).
    cout << Teste::n << endl; // 6 = 1+5.
    return 0;
}
```

- A saída é 7 e 6

Membros estáticos

- Portanto, membros estáticos têm a mesma caraterística de variáveis globais, porém dentro do escopo da classe.

- Por esta razão, e para evitar que sejam declarados várias vezes, somente podemos incluir o seu protótipo (ou declaração) na especificação da classe, mas jamais a sua definição (ou inicialização).

- Para inicializar um atributo estático devemos incluir uma inicialização explícita fora da classe, com escopo global como no exemplo anterior:

```
int Teste::n=0;
```

Membros estáticos

➔ Como tem-se o mesmo valor para todos os objetos daquela classe, podemos acessá-lo por meio de um objeto da classe ou diretamente usando o nome da classe e o operador de escopo (::). Evidentemente que isto só é válido para membros estáticos:

```
cout << a.n;        // usando um objeto da classe
cout << Teste::n;   // usando o nome da classe
```

➔ As duas linhas acima fazem referência à mesma variável: a variável estática *n* dentro da classe *Teste*, que é portanto compartilhada por todos os objetos da classe.

➔Novamente, lembre-se de que *Teste*::*n* é como uma variável global. A única diferença é o seu nome (pois foi definida dentro da classe) e a possibilidade de fazer restrições de acesso fora da classe (declarando numa seção *private* ou *protected*).

Membros estáticos

- Da mesma forma que podemos incluir atributos estáticos numa classe, também podemos incluir *métodos estáticos*.

- Estes métodos são como funções globais que são chamadas como se fossem métodos de uma dada classe, mas sem a necessidade de instanciar um objeto.

- Métodos estáticos só podem fazer referência a membros estáticos, jamais a membros não estáticos. Também não podem usar a palavra-chave *this*, uma vez que ela representa um ponteiro para um objeto.

- Métodos estáticos não são métodos de um objeto, mas sim membros da classe.

Membros estáticos

- Exemplo de uso de métodos estáticos.

```
class Math {
  public:
    static float soma(float a, float b)
      {return a+b;}
    static float subtrai(float a, float b)
      {return a-b;}
    static float multiplica(float a, float b)
      {return a*b;}
    static float divide(float a, float b)
     {return a/b;}
};
...
```

Membros estáticos

```
...
int main() {
  float x = Math::soma(4, 5);
  float y = Math::divide(20, 4);
  float z;
  Math m;

  z = m.subtrai(6, 2); // isto foi só para mostrar
                       // que podemos executar
                       // normalmente através do
                       // objeto também.

  cout << "X: " << x << ", Y: " << y << endl;
  return 0;
}
```

Métodos *inline*

Métodos *inline*

➔ Um *método inline* é aquele que o compilador não gera código para a chamada do método e sim insere o código completo do método em todo lugar que é invocado.

➔ A razão para o uso de métodos *inline* é eficiência. O tempo consumido pela preparação da chamada de um método (criação e empilhamento dos argumentos, armazenamento do endereço de retorno, desvio para o código da rotina) é eliminado.

➔ No entanto, o tamanho final do código aumenta, uma vez que o código do método é inserido em toda parte do programa que o invoca.

Métodos *inline*

➔ Os métodos cujo corpo é definido dentro da declaração da classe são, por padrão, *inline*. No entanto, é prática não declarar métodos maiores que uma linha dentro da classe para não tornar a especificação da mesma confusa.

➔ Como fazer então se queremos declarar um método *inline* fora da declaração da classe? Neste caso usamos a palavra-chave *inline* na frente da declaração do método, como mostrado abaixo:

```
class X {
   ...
   f();
};

inline X::f() { ...
```

Introdução a diagrama de classes

Introdução a diagrama de classes

- É possível registrar diretamente em uma linguagem de programação os objetos percebidos em uma determinada realidade. No entanto, é melhor utilizarmos uma *notação gráfica* intermediária para melhor visualizarmos os objetos e as relações entre objetos e ir alterando esta representação até estarmos seguros que possuímos um entendimento razoável do problema. A partir daí sim, iniciamos a codificação da solução.

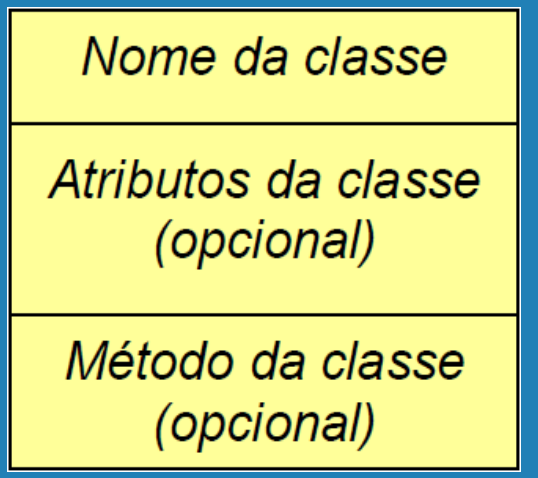
- A notação gráfica que mostraremos denota as relações estáticas entre classes de objetos. Ela faz parte do conjunto de notações da UML (Unified Modeling Language ou Linguagem de Modelagem Unificada) proposta por Grady Booch, James Rumbaugh e Ivar Jacobson em 1995.

Introdução a diagrama de classes

➔ O *Diagrama de Classes* representa graficamente as classes do sistema e o relacionamento estático entre as classes, isto é, o relacionamento que não muda com o tempo. Por exemplo, em um sistema acadêmico, um aluno cursa várias disciplinas. O número de disciplinas e a disciplina que efetivamente está sendo cursada pode alterar, mas o vínculo aluno-cursa-disciplinas permanece.

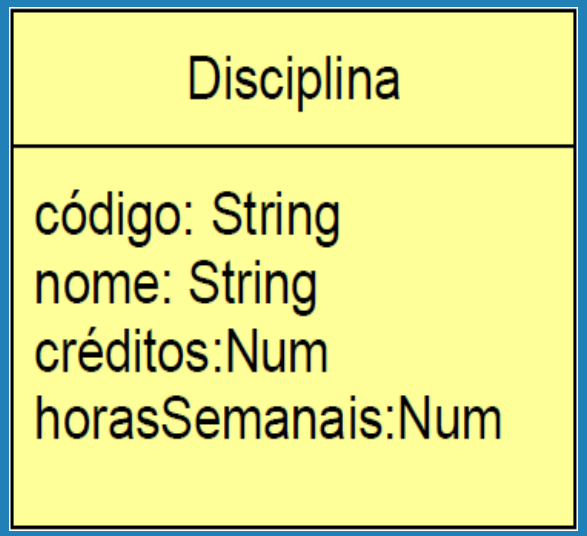
➔ Uma classe é representada no diagrama de classes por meio de um retângulo, que pode ser dividido em até três seções horizontais, como mostrado na figura ao lado:

- A seção superior é usada para registrar o nome da classe.
- A seção intermediária é reservada para registro das propriedades da classe, caso existam.
- Na seção inferior é registrada a interface dos métodos que pertencem à classe, caso existam.



Introdução a diagrama de classes

➔ De modo geral, por razões de simplicidade, não se representam os métodos que tratam da alteração dos atributos ou que criam a classe. Assume-se que toda classe possui tais métodos. Portanto, podemos simplificar a representação vista, omitindo a última seção, se só houver métodos de alteração e de inicialização/finalização.

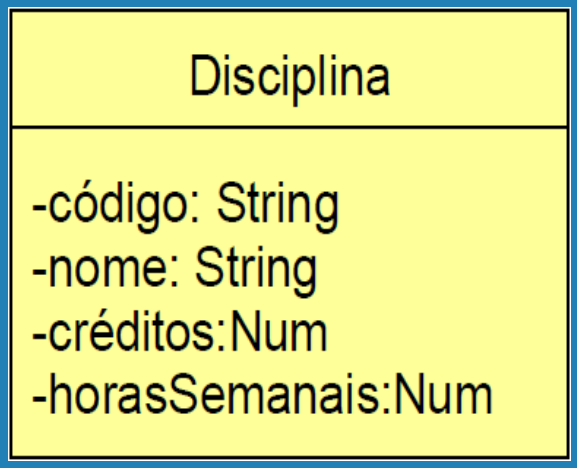


Introdução a diagrama de classes

➔ Nos diagramas, pode-se indicar a visibilidade dos membros da classe. As visibilidades possíveis, juntamente com os símbolos adotados estão listados na tabela abaixo:

Visibilidade	*Símbolo*	*Descrição*
Pública	+	Sem restrição de acesso.
Protegida	#	Pode ser acessado apenas na própria classe e por subclasses.
Privada	-	Pode ser acessado apenas na própria classe.

➔ A visibilidade é definida para um atributo ou método precedendo a declaração com o símbolo adequado, como na figura ao lado (uso do '-' para indicar *private*):



Introdução a diagrama de classes

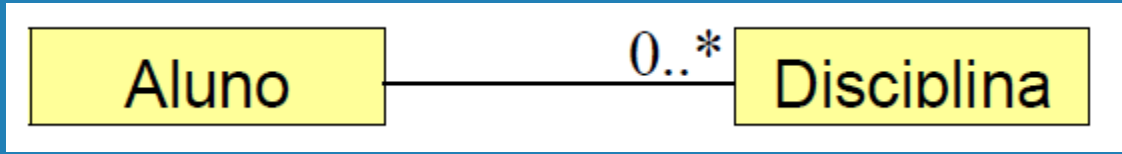
- As classes podem se relacionar de diversas formas:
  - por associação comum
  - por agregação
  - e por generalização

- A seguir é apresentada cada forma de relacionamento, juntamente com suas notações.

- *Associação comum*: a notação utilizada para associar duas classes é simplesmente uma linha unindo-as:

Introdução a diagrama de classes

- A figura anterior expressa que alunos se associam com disciplinas mas não indica se um aluno se relaciona com várias ou apenas uma disciplina. Esta informação é chamada de *cardinalidade da relação* e é expressada anotando-se o valor da cardinalidade na associação junto à classe que está sendo relacionada:

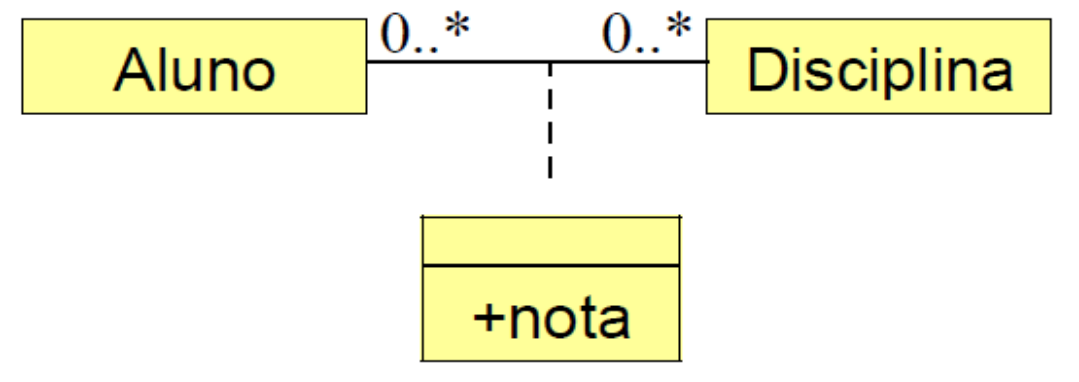


- Algumas representações de cardinalidade:

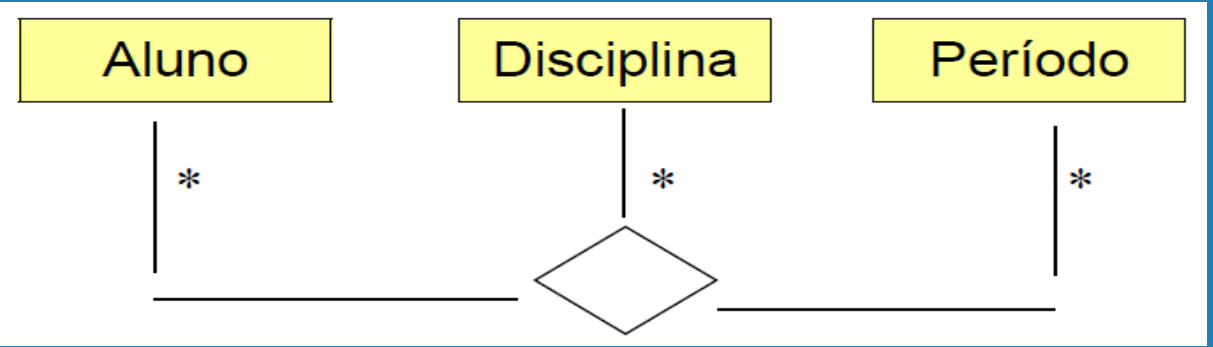
Notação	*Descrição*
1	Exatamente um
* ou 0..*	Zero ou mais
0..1	Opcional (zero ou um)
n..m	Máximo e mínimo

Introdução a diagrama de classes

- Uma associação pode ter atributos próprios, ou seja, atributos que não pertençam a nenhuma das classes envolvidas na associação mas sim à própria associação:

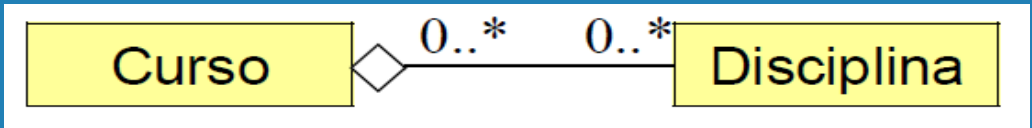


- Nada impede que mais de duas classes participem de uma associação:

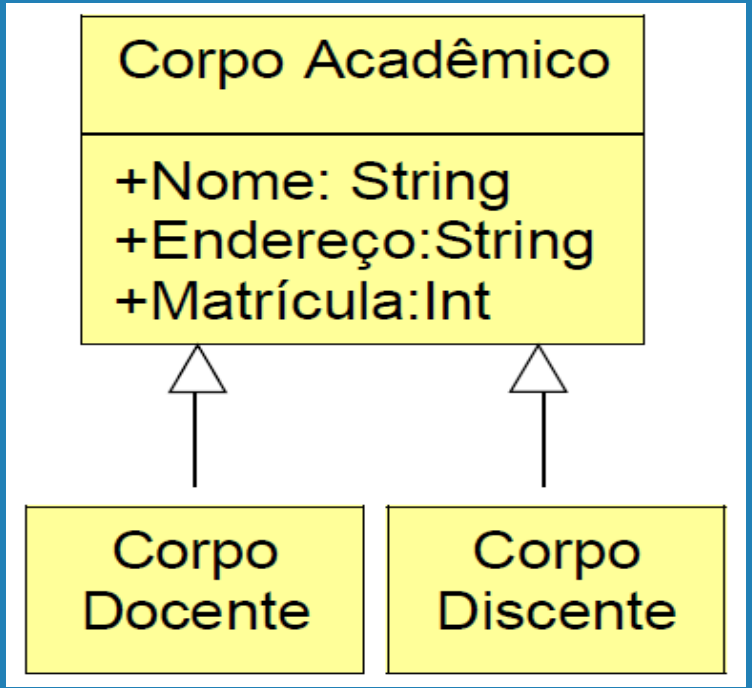


Introdução a diagrama de classes

➔ *Agregação*: alguns objetos são compostos por outros objetos. Por exemplo, um carro é composto por chassi, lataria, pneus e motor. Este, por sua vez, é composto por carburador, pistões, bloco, etc. Este tipo de associação é representada por uma linha com um losango na ponta.

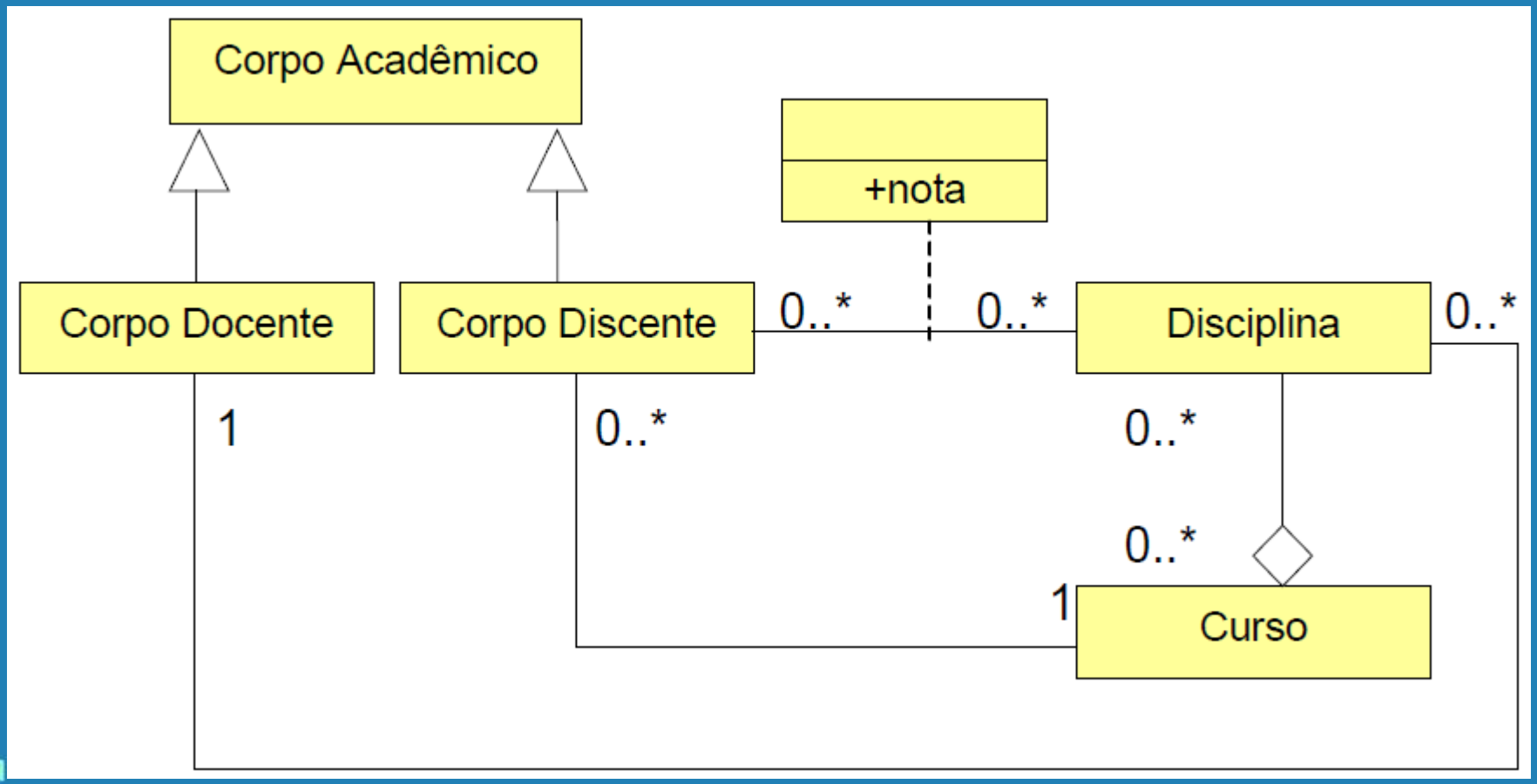


➔ *Generalização*: O último tipo de associação entre classes é o que ocorre entre superclasses e subclasses. Uma superclasse é uma generalização das suas subclasses, que herdam os atributos e métodos da primeira. A notação utilizada para representar a generalização é uma linha com um triângulo na extremidade da classe mais genérica.



Introdução a diagrama de classes

- Vejamos um exemplo um pouco mais completo:



Introdução a diagrama de classes

➔ Uma vez que o diagrama de classes esteja amadurecido, parte-se então para a programação. A regra geral para a implementação do que está representado no digrama é a seguinte:

- Cada classe e associação com atributos gera uma classe no programa.
- Cada associação pode ser implementada, por exemplo, com arranjos ou ponteiros.
- As generalizações são implementadas com o uso de herança da linguagem em questão.