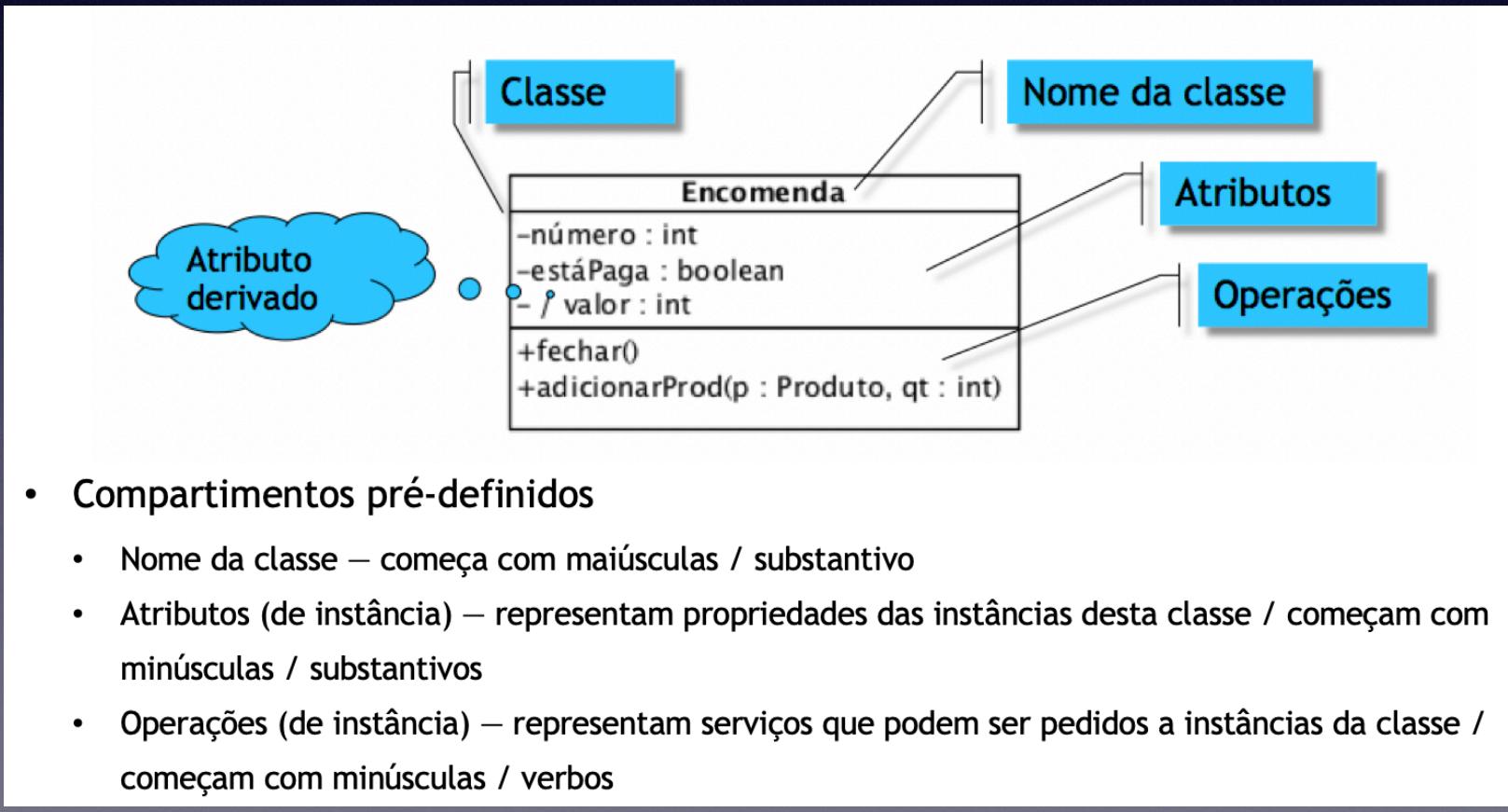


# A classe Aluno

- Comecemos por pensar primeiro na arquitectura da classe
  - antes de codificar devemos “modelar” o estado e o comportamento dos objectos
  - vamos recorrer a uma versão simplificada de uma linguagem de modelação, a **UML**
  - modelamos estado, comportamento e visibilidade de v.i. e métodos de instância
  - modelamos o relacionamento entre classes

# Breve introdução ao Diagrama Classes UML

- com excertos dos slides de José Campos  
(copyrigth UC de DSS)



# Visibilidade de atributos e operações

- O nível de visibilidade (acesso) que se pretende para cada atributo/operação é representado com as seguintes anotações:
  - privado – só acessível ao objecto a que pertence (cf. encapsulamento)
  - # protegido – acessível a instâncias das sub-classes (atenção: em Java fica também acessível a instâncias de classes do mesmo *package*!)
  - pacote/*package* – acessível a instâncias de classes do mesmo *package* (nível de acesso por omissão)
  - + público – acessível a todos os objectos no sistema (que conhecem o objecto a que o atributo/operação pertence!)

# Declaração de atributos

- Atributos

**«esterótipo» visibilidade / nome : tipo [multiplicidade] = valorInic {propriedades}**

- Exemplos

morada

- morada= “Braga” {addedBy=“jfc”, date=“18/11/2011”}

- morada: String [1..2] {leaf, addOnly, addedBy=“jfc”}

Só o nome é obrigatório!



# Declaração de operações

- Operações

Obrigatório!

in | out | inout | return

«esterótipo» visibilidade nome (direção nomeParam : tipo = valorOmiss) : tipo

{propriedades}

- Exemplos

setNome

por omissão é “in”

```
+ setNome(nome = “SCX”) {abstract}  
+ getName() : String {isQuery, risco = baixo}  
# getName(out nome) {isQuery}  
+ «create» Pessoa()
```

in - parâmetro de entrada  
out - parâmetro de saída  
inout - parâmetro de entrada/saída  
return - operação retorna o parâmetro como um dos seus valores de retorno

# A classe Aluno: modelo

```
a          Aluno
+numero : String
-nota : int
-nome : String
-curso : String
+Aluno()
+Aluno(numero : String, nota : int, nome : String, curso : String)
+Aluno(umAluno : Aluno)
+getNumero() : String
+getNota() : int
+getNome() : String
+getCurso() : String
+setNota(novaNota : int) : void
+setNumero(numero : String) : void
+setNome(nome : String) : void
+setCurso(curso : String) : void
+toString() : String
>equals(o : Object) : boolean
+clone() : Object
```

variáveis de  
instância

construtores

+

métodos de  
instância

- esta abordagem permite-nos:
  - pensar na classe antes de a implementar
  - comunicar qual é o comportamento esperado
  - determinar quais são, e de que tipo, os parâmetros dos métodos
  - permite ao resto da equipa começar a trabalhar noutras classes sabendo qual é a estrutura da classe Aluno.

# A classe Aluno: implementação

- declaração das variáveis de instância

```
/**  
 * Classe Aluno.  
 * Classe que modela de forma muito simples a  
 * informação e comportamento relevante de um aluno.  
 *  
 * @author MaterialP00  
 * @version 20200216  
 * @version 20240219  
 */  
public class Aluno {  
  
    private String numero;  
    private int nota;  
    private String nome;  
    private String curso;
```

- construtores: vazio, parametrizado e de cópia

```
public Aluno() {  
    this.numero = "";  
    this.nota = 0;  
    this.nome = "";  
    this.curso = "";  
}  
  
public Aluno(String numero, int nota, String nome, String curso) {  
    this.numero = numero;  
    this.nota = nota;  
    this.nome = nome;  
    this.curso = curso;  
}  
  
public Aluno(Aluno umAluno) {  
    this.numero = umAluno.getNumero();  
    this.nota = umAluno.getNota();  
    this.nome = umAluno.getNome();  
    this.curso = umAluno.getCurso();  
}
```

## ● métodos *getters* e *setters*

```
/**  
 * Método que devolve o número de um aluno.  
 *  
 * @return String com o número do aluno  
 */  
public String getNumero() {  
    return this.numero;  
}  
  
/**  
 * Método que devolve a nota de um aluno.  
 *  
 * @return int com o número do aluno  
 */  
public int getNota() {  
    return this.nota;  
}  
  
/**  
 * Método que devolve o nome de um aluno.  
 *  
 * @return String com o nome do aluno  
 */  
public String getNome() {  
    return this.nome;  
}  
  
/**  
 * Método que devolve o curso de um aluno.  
 *  
 * @return String com o número do aluno  
 */  
public String getCurso() {  
    return this.curso;  
}
```

```
public void setNota(int novaNota) {  
    this.nota = novaNota;  
}
```

```
public void setNumero(String numero) {  
    this.numero = numero;  
}
```

```
public void setNome(String nome) {  
    this.nome = nome;  
}
```

```
public void setCurso(String curso) {  
    this.curso = curso;  
}
```

Classe Aluno. Classe que modela de forma muito simples a informação e comportamento relevante de um aluno.

Version:

20200216

Author:

MaterialPOO

### Constructor Summary

Constructors	Description
<code>Aluno()</code>	Constructores para a classe Aluno
<code>Aluno(Aluno umAluno)</code>	
<code>Aluno(java.lang.String numero, int nota, java.lang.String nome, java.lang.String curso)</code>	

### Method Summary

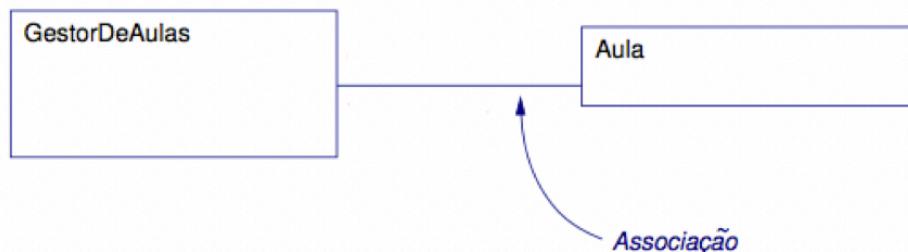
All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method	Description
<code>Aluno</code>	<code>clone()</code>	Implementação do método de clonagem de um Aluno
<code>boolean</code>	<code>equals(java.lang.Object o)</code>	Implementação do método de igualdade entre dois Aluno Redefinição do método equals de Object.
<code>java.lang.String</code>	<code>getCurso()</code>	Método que devolve o curso de um aluno.
<code>java.lang.String</code>	<code>getNome()</code>	Método que devolve o nome de um aluno.
<code>int</code>	<code>getNota()</code>	Método que devolve a nota de um aluno.
<code>java.lang.String</code>	<code>getNumero()</code>	Método que devolve o número de um aluno.
<code>void</code>	<code>setCurso(java.lang.String curso)</code>	
<code>void</code>	<code>setNome(java.lang.String nome)</code>	
<code>void</code>	<code>setNota(int novaNota)</code>	
<code>void</code>	<code>setNumero(java.lang.String numero)</code>	
<code>java.lang.String</code>	<code>toString()</code>	Implementação do método toString comum na maioria das classes Java.

# UML: Relação entre classes - dependência



- Indica que a definição de uma classe está dependente da definição de outra.
- Utiliza-se normalmente para mostrar que instâncias da origem utilizam, de alguma forma, instâncias do destino (por exemplo: um parâmetro de um método)
- Uma alteração no destino (quem é usado) pode alterar a origem (quem usa)

# Relações entre classes - Associação

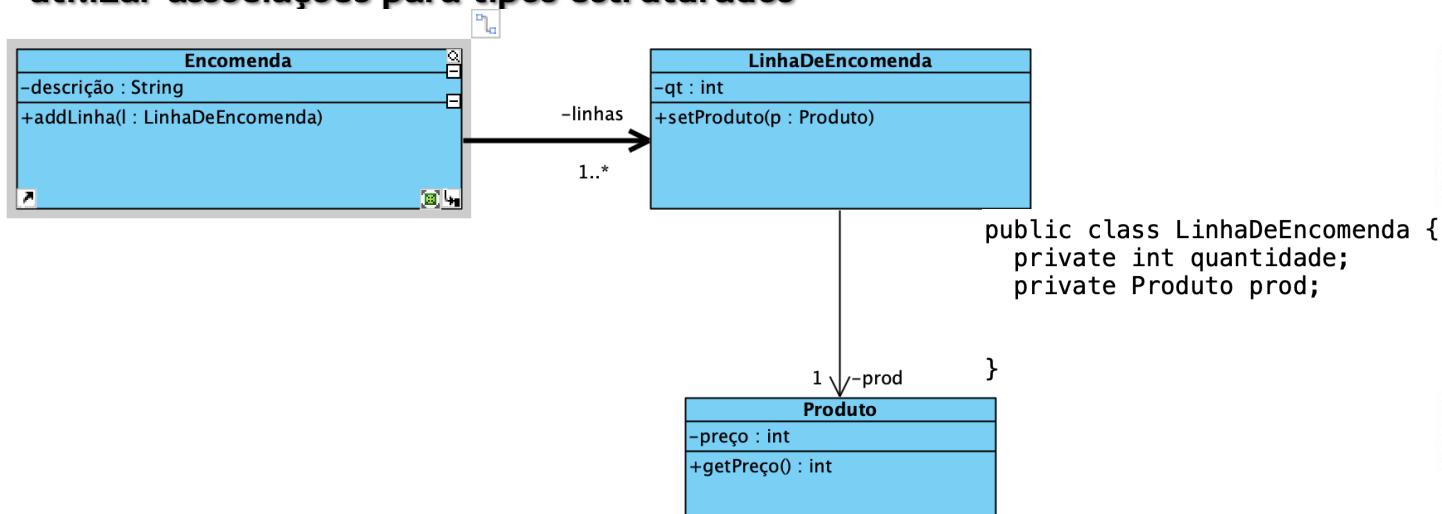


- Indica que objectos de uma estão ligados a objectos de outra – define uma relação entre os objectos
- Noção de naveabilidade (cf. diagramas E-R)
- Por omissão representam navegação bidireccional – mas pode indicar-se explicitamente o sentido da naveabilidade.



# Associações vs Atributos

- Atributos (de instância) representam propriedades das instâncias das classes
  - São codificados como variáveis de instância
- Associações também representam propriedades das instâncias das classes
  - também são codificados como variáveis de instância
- Atributos devem ter tipos simples
  - **utilizar associações para tipos estruturados**

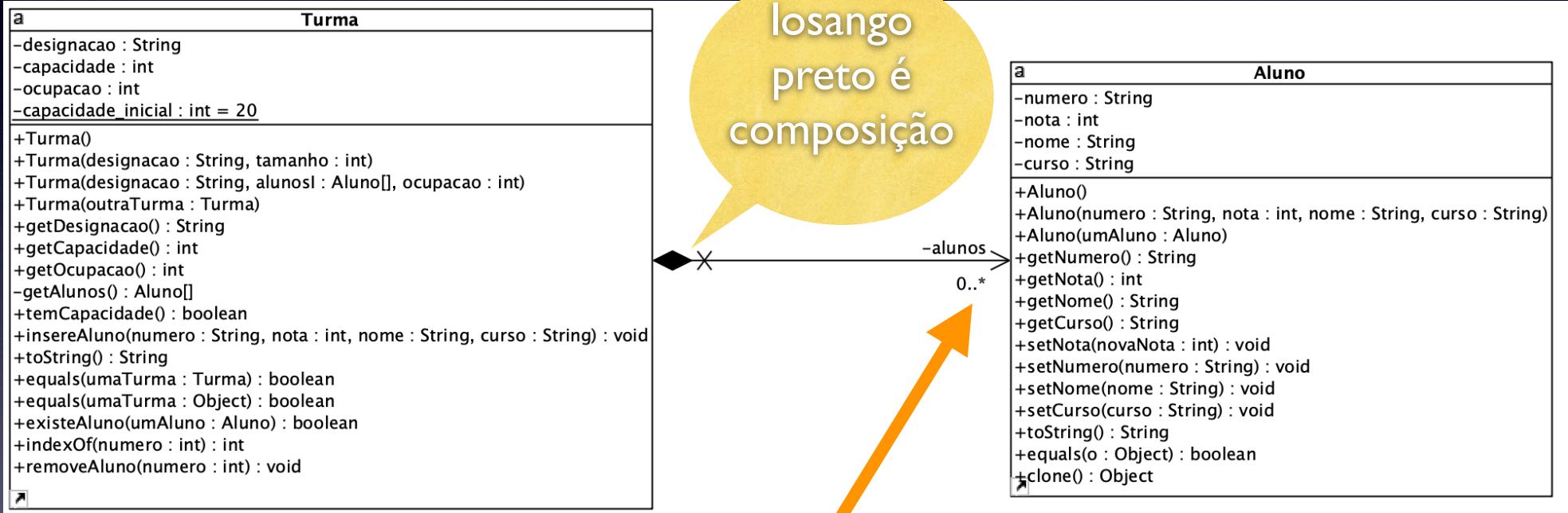


# A classe Turma

- criação de um objecto que permita guardar instâncias de Aluno
- como estrutura de dados vamos utilizar um array de objectos do tipo Aluno
  - Aluno alunos[ ]
- A utilização de Aluno na definição de Turma corresponde à utilização de **composição** na definição de objectos mais complexos

- às situações em que uma classe seja composta por outros objectos e:
  - faça a gestão do ciclo de vida dos mesmos
  - faça a criação dos objectos internamente
  - não os receba por parâmetro já criados
  - vamos designar por **composição** e vamos, nessa situação, respeitar escrupulosamente o encapsulamento!

# A classe Turma: modelo



a var. de instância chama-se alunos, é privada e pode ter zero ou mais instâncias de Aluno

## Constructors

### Constructor

`Turma()` Constructor por omissão (vazio) para objectos da classe Turma

`Turma(java.lang.String designacao, int tamanho)` Constructor parametrizado de Turma.

`Turma(java.lang.String designacao, Aluno[] alunosI, int ocupacao)` Constructor parametrizado de Turma em que se envia já os alunos que fazem parte da turma.

`Turma(Turma outraTurma)` Constructor de cópia de Turma.

## Method Summary

### All Methods    Instance Methods    Concrete Methods

Modifier and Type	Method	Description
boolean	<code>equals(java.lang.Object umaTurma)</code>	Método equals standard do Java.
boolean	<code>equals(Turma umaTurma)</code>	Método equals.
boolean	<code>existeAluno(Aluno umAluno)</code>	De acordo com o funcionamento tipo destes métodos, vai-se percorrer o array e enviar o método equals a cada objecto
int	<code>getCapacidade()</code>	
java.lang.String	<code>getDesignacao()</code>	
int	<code>getOcupacao()</code>	
int	<code>indexOf(int numero)</code>	Método que percorre o array e dá a posição em que se encontra determinado aluno.
void	<code>insereAluno(Aluno umAluno)</code>	Método que insere um aluno na turma, mas recebe já uma instância da classe Aluno.
void	<code>insereAluno(java.lang.String numero, int nota, java.lang.String nome, java.lang.String curso)</code>	Este método assume que se verifique previamente se ainda existe espaço para mais um aluno na turma.
void	<code>removeAluno(int numero)</code>	Método que remove um elemento do array.
boolean	<code>temCapacidade()</code>	
java.lang.String	<code>toString()</code>	Método toString por questões de compatibilização com as restantes classes do Java.

- declaração das v.i.

```
/**  
 * Primeira implementação de uma turma de alunos.  
 * Assume que a turma é mantida num array.  
 *  
 * @author MaterialPOO  
 * @version 20240220  
 */  
public class Turma {  
    private String designacao;  
    private Aluno[] alunos;  
    private int capacidade;  
  
    //variaveis internas para controlo do numero de alunos  
    private int ocupacao;  
  
    //se não for especificado o tamanho da turma usa-se esta constante  
    private static final int capacidade_inicial = 20;
```

## ● construtores

```
* Constructor for objects of class Turma
*/
public Turma() {
    this.designacao = new String();
    this.alunos = new Aluno[capacidade_inicial];
    this.capacidade = capacidade_inicial;
    this.ocupacao = 0;
}

public Turma(String designacao, int tamanho) {
    this.designacao = designacao;
    this.alunos = new Aluno[tamanho];
    this.capacidade = tamanho;
    this.ocupacao = 0;
}

public Turma(Turma outraTurma) {
    this.designacao = outraTurma.getDesignacao();
    this.capacidade = outraTurma.getCapacidade();
    this.ocupacao = outraTurma.getOcupacao();
    this.alunos = outraTurma.getAlunos();
}
```

- **getters**

```
public String getDesignacao() {  
    return this.designacao;  
}  
  
public int getCapacidade() {  
    return this.capacidade;  
}  
  
public int getOcupacao() {  
    return this.ocupaçao;  
}  
  
/**  
 * Método privado (auxiliar)  
 * Possível problema de encapsulamento ao partilhar  
 * o endereço do array.  
 *  
 * @return Array com os objectos do tipo Aluno  
 */  
private Aluno[ ] getAlunos() {  
    return this.alunos;  
}
```

- o método `getAlunos` é declarado como auxiliar e privado. Porquê?

- inserir um novo Aluno

```
/**  
 * Este método assume que se verifique previamente se  
 * ainda existe espaço para mais um aluno na turma.  
 *  
 * Em futuras versões desta classe poderemos fazer internamente a  
 * gestão das situações de erro. Neste momento assume-se que a  
 * pré-condição é verdadeira.  
 */  
  
public void insereAluno(String numero, int nota, String nome, String curso) {  
    this.alunos[this.ocupaçao] = new Aluno(numero, nota, nome, curso); //encapsulamento garantido  
    this.ocupaçao++;  
}
```

- quem cria a instância de Aluno é a classe Turma

- Podemos criar um método para as situações em que o objecto Aluno é criado fora da Turma:

```
/**  
 * Método que insere um aluno na turma, mas recebe já uma instância da  
 * classe Aluno.  
 * Como forma de garantir o encapsulamento cria-se uma cópia do objecto recebido.  
 */  
  
public void insereAluno(Aluno umAluno) {  
    this.alunos[this.ocupaçao] = new Aluno(umAluno);  
    this.ocupaçao++;  
}
```

- Como foi decidido, na fase de concepção, que a estratégia de associação previa uma composição então é necessário explicitamente clonar o objecto.

# O método clone

- este método tem como objectivo a criação de uma cópia do objecto a quem é enviado
  - a noção de cópia depende muito da classe que faz a implementação
  - a noção geral é que `x.clone() != x`
  - sendo que,

`x.clone().getClass() == x.getClass()`

# O método clone

- regra geral, e de acordo com a visão em POO, a expressão seguinte deve prevalecer
$$x.clone().equals(x),$$
- embora isso dependa muito da forma como ambos os métodos estão implementados
- a implementação de clone é relativamente simples

# O método clone

- na metodologia de POO já temos um método que faz cópia de objectos
  - o construtor de cópia de cada classe
- Dessa forma podemos dizer que apenas temos de invocar esse construtor e passar-lhe como referência o objecto que recebe a mensagem - neste caso o *this*

# O método clone

- implementação do método clone da classe Aluno

```
/**  
 * Implementação do método de clonagem de um Aluno  
 *  
 * @return objecto do tipo Aluno  
 */  
  
public Aluno clone() {  
    return new Aluno(this);  
}
```

- optamos por devolver um objecto do mesmo tipo de dados e não Object como é a norma do clone em Java.

# Clone vs Encapsulamento

- a utilização de `clone()` permite que seja possível preservarmos o encapsulamento dos objectos, desde que:
  - seja feita uma cópia dos objectos à entrada dos métodos
  - seja devolvida uma cópia dos objectos e não o apontador para os mesmos

# A clonagem de objectos

- Duas abordagens:
  - *shallow clone*: cópia parcial que deixa endereços partilhados (cria as estruturas de dados mas partilha os conteúdos)
  - *deep clone*: cópia em que nenhum objecto partilha endereços com outro

- A sugestão é utilizar, se tivermos modelado uma composição, *deep* clone, na medida em que podemos controlar todo o processo de acesso aos dados
- **REGRA:** clone do objecto = “soma” do clone de todas as suas variáveis de instância
  - tipos simples e objectos imutáveis (String, Integer, Float, etc.) não precisam (não devem!) ser clonados.

- A saber:
  - implementar o clone como sendo uma invocação do construtor de cópia
  - o método `clone()` existente nas classes Java é sempre *shallow*, e devolve sempre um `Object` (se usado, é necessário fazer cast)
  - os clones que vamos fazer, nas nossas classes, devolvem sempre um tipo de dados da classe