

sobre classes e instâncias

“Classes and object are separate yet intimately related concepts. Specifically, every object is the instance of some class, and every class has zero or more instances. For practically all applications, classes are static; therefore, their existence, semantics, and relationships are fixed prior to the execution of a program. Similarly, the class of most objects is static, meaning that once an object is created, its class is fixed. In sharp contrast, however, objects are typically created and destroyed at a furious rate during the lifetime of an application.”

Definição do objecto classe

Estado

identificação das variáveis de instância

Comportamento

construtores/destrutores

getters e setters

outros métodos de instância, decorrentes
do que representam

A referência *this*

é usual precisarmos de referenciar o objecto que recebe a mensagem

mas, no contexto da escrita do código da classe, ainda não sabemos como é que se vai chamar o objecto

sempre que precisamos de ter acesso a uma variável do objecto podemos usar a referência *this*

uma utilização normal é querermos desambiguar e identificar as variáveis de instância

Por exemplo, em

```
/**  
 * Construtor parametrizado de Ponto.  
 * Aceita como parâmetros os valores para cada coordenada.  
 */  
public Ponto(int x, int y) {  
    this.x = x;  
    this.y = y;  
}
```

a utilização de *this* permite desambiguar a qual das variáveis nos estamos a referir

a referência *this* pode ser utilizada para identificar um outro método da instância

```
public void movePonto(int cx, int cy) {  
    this.setX(cx);  
    this.setY(cy);  
}
```

no caso de termos escrito apenas `setX(cx)`
o compilador teria acrescentado
automaticamente a referência *this*

também utilizada para invocar os construtores
(dentro de outros construtores)

Regras de acesso a variáveis e métodos

a declaração deve ser complementada com informação sobre o nível de visibilidade das variáveis e métodos.

Tipo de Modificador	Visibilidade no código
public	a partir de qualquer classe
private	apenas acessível dentro da classe
protected	acessível a partir da classe, de classes do mesmo package e de todas as <u>subclasses</u>
default	acessível a partir da classe e classes do mesmo package

para garantir o total encapsulamento do objecto as variáveis de instância devem ser declaradas como **private**

ao ter encapsulamento total é necessário garantir que existem métodos que permitem o acesso e modificação das variáveis de instância.

os métodos que se pretendem que sejam visíveis do exterior devem ser declarados como **public**

A classe Aluno

Comecemos por pensar primeiro na arquitectura da classe

antes de codificar devemos “modelar” o estado e o comportamento dos objectos

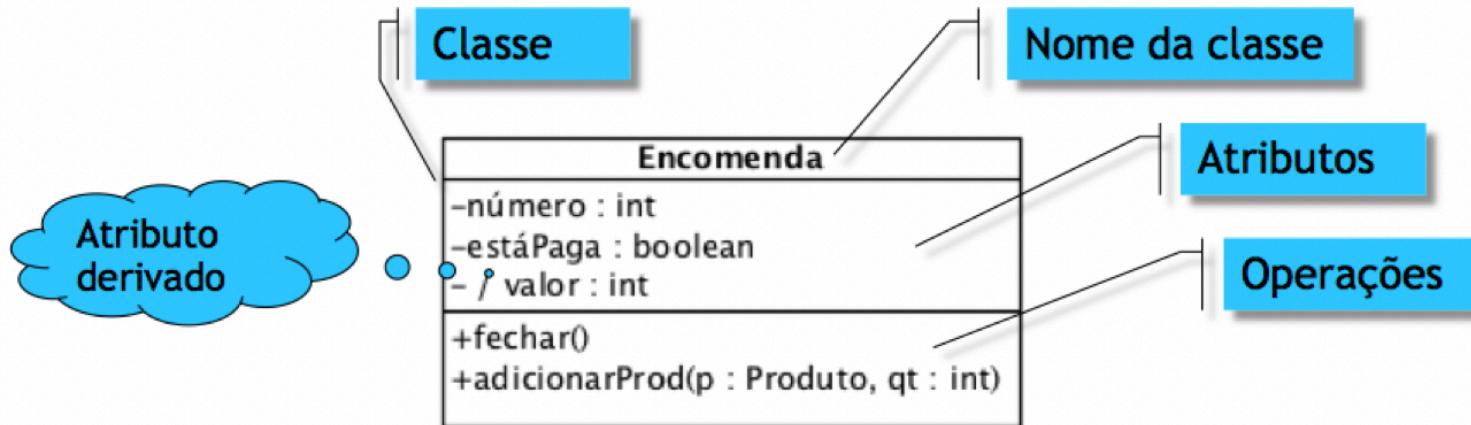
vamos recorrer a uma versão simplificada de uma linguagem de modelação, a **UML**

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 DSS 19/20)



- Compartimentos pré-definidos
 - Nome da classe – começa com maiúsculas / substantivo
 - Atributos (de instância) – representam propriedades das instâncias desta classe / começam com minúsculas / substantivos
 - Operações (de instância) – representam serviços que podem ser pedidos a instâncias da classe / começam com minúsculas / verbos

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 conhecam 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

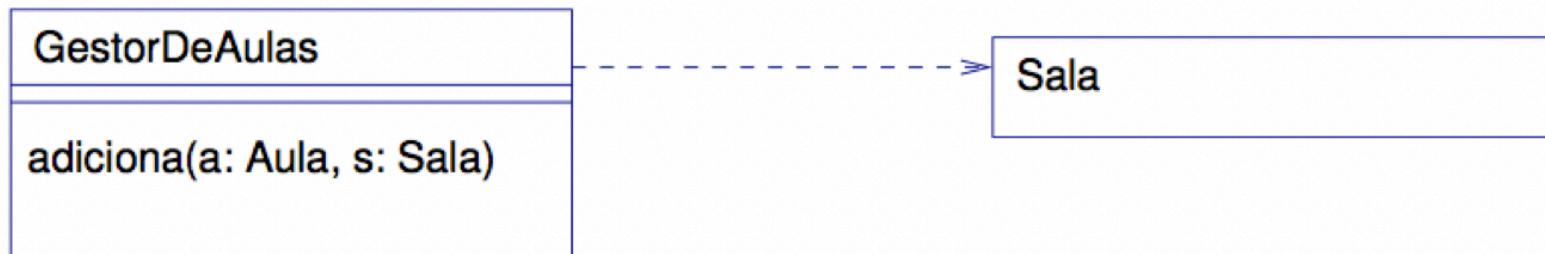
setNome

por omissão é “in”

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

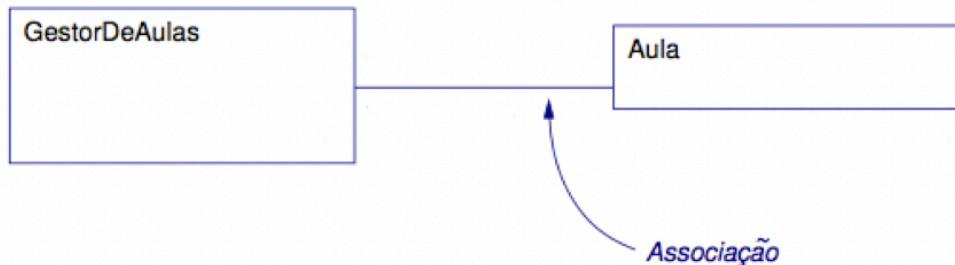
in - parâmetro de entrada
out - parâmetro de saída
inout - parâmetro de entrada/saída
return - operação retorna o parâmetro como um dos seus valores de retorno

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

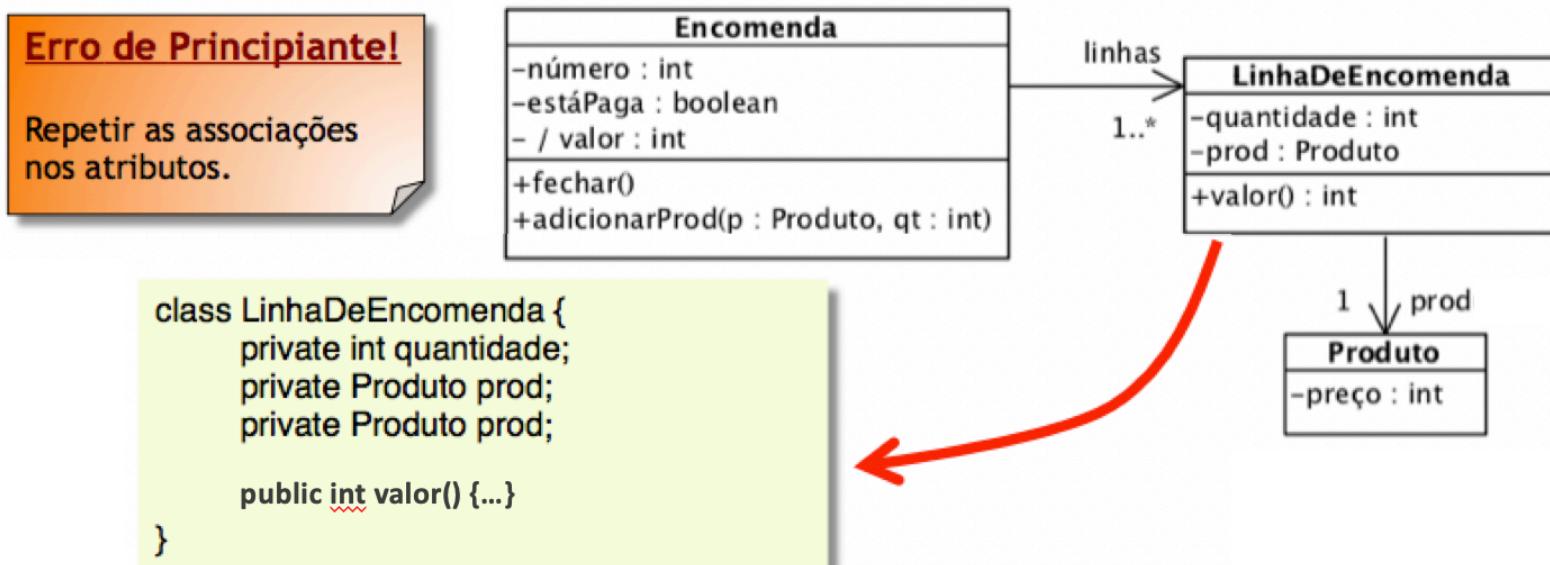


- Indica que objectos de uma estão ligados a objectos de outra – define uma relação entre os objectos
- Noção de naveabilidade (cf. diagramas E-R)
- Por omissão representam navegação bidireccional – mas pode indicar-se explicitamente o sentido da naveabilidade.



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 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

permitem 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  
 */  
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 o nome de um aluno.  
 *  
 * @return String com o nome do aluno  
 */  
public String getNome() {  
    return this.nome;  
}
```

```
/**  
 * 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 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

Constructor

Description

`Aluno ()`

Constructores para a classe Aluno

`Aluno (Aluno umAluno)`

`Aluno (java.lang.String numero, int nota, java.lang.String nome, java.lang.String curso)`

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.

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

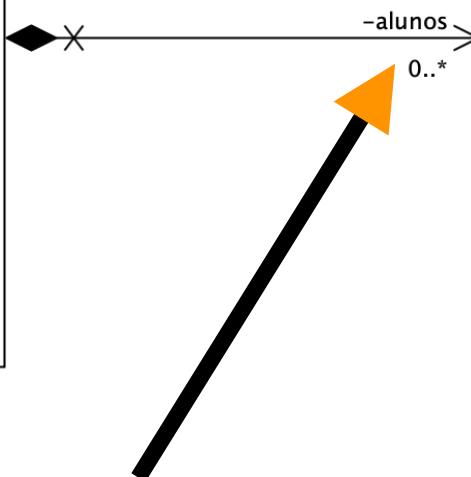
vamos designar por **composição** e vamos, nessa situação, respeitar escrupulosamente o encapsulamento!

A classe Turma: modelo

```
a           Turma
-----+
-designacao : String
-capacidade : int
-ocupacao : int
-capacidade_inicial : int = 20
-----+
+Turma()
+Turma(designacao : String, tamanho : int)
+Turma(designacao : String, alunos : Aluno[], ocupacao : int)
+Turma(outraTurma : Turma)
+getDesignacao() : String
+getCapacidade() : int
+getOcupacao() : int
+getAlunos() : Aluno[]
+temCapacidade() : boolean
+insereAluno(numero : String, nota : int, nome : String, curso : String) : void
+toString() : String
>equals(umaTurma : Turma) : boolean
>equals(umaTurma : Object) : boolean
+existeAluno(umAluno : Aluno) : boolean
+indexOf(numero : int) : int
+removeAluno(numero : int) : void

```

losango
preto é
composição



```
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

```

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

decla o das v.i.

```
/**  
 * Primeira implementa o de uma turma de alunos.  
 * Assume que a turma   mantida num array.  
 *  
 * @author MaterialP00  
 * @version 20200216  
 */  
public class Turma {  
    private String designacao;  
    private Aluno[ ] alunos;  
    private int capacidade;  
  
    //variaveis internas para controlo do numero de alunos  
    private int ocupacao;  
  
    //se n o 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.ocupaao;  
}
```

```
/**  
 * 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 auxiliar e privado

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.  
 *  
 * Este método deverá ser reescrito em futuras implementações  
 * para evitar potenciais quebras de encapsulamento - já feito ao  
 * agregar uma cópia do aluno passado como parâmetro.  
 *  
 */
```

```
public void insereAluno(String numero, int nota, String nome, String curso) {  
    this.alunos[this.ocupaçao] = new Aluno(numero, nota, nome, curso); //encapsulamento  
    this.ocupaçao++;  
}
```

quem cria a instância de Aluno é a classe Turma

caso, o método tivesse a assinatura

```
public void insereAluno(Aluno umAluno)
```

seria necessário que a classe fizesse uma cópia do objecto, para garantir que não se partilha o apontador.

Como foi decidido, na fase de concepção, que a arquitectura previa uma composição (losango preto) 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.

Constructor Summary

Constructors

Constructor	Description
<code>Turma()</code>	Constructor for objects of class Turma
<code>Turma(java.lang.String designacao, int tamanho)</code>	
<code>Turma(java.lang.String designacao, Aluno[] alunosI, int ocupacao)</code>	
<code>Turma(Turma outraTurma)</code>	

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(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.

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

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

Igualdade de objectos

Como implementar os métodos

```
public boolean existeAluno(Aluno a)
```

```
public void removeAluno(Aluno a)
```

como é que determinamos se o objecto
está efectivamente dentro do array de
alunos?

A solução

`alunos[i] == a`, não é eficaz porque compara os apontadores (e pode ter havido previamente um clone)

`(alunos[i]).getNumero() == a.getNumero()`, assume demasiado sobre a forma como se comparam alunos

Quem é a melhor entidade para determinar como é que se comparam objectos do tipo Aluno?

através da disponibilização de um método, na classe Aluno, que permita comparar instâncias de alunos

é importante que esse método seja universal, isto é, que tenha sempre a mesma assinatura

é importante que todos os objectos respondam a este método

public boolean equals(Object o)

dessa forma o método existeAluno(Aluno a) da classe Turma, assume a seguinte forma:

```
public boolean existeAluno(Aluno umAluno) {  
    boolean resultado = false;  
  
    if (umAluno != null) {  
  
        for(int i=0; i< this.ocupaçao && !resultado; i++)  
            resultado = this.alunos[i].equals(umAluno);  
  
        return resultado;  
    }  
    else  
        return false;  
}
```

Em resumo:

método de igualdade é determinante para que seja possível ter colecções de objectos

o método de igualdade não pode ser codificado a não ser pela classe:
abstração de dados

existem um conjunto de regras básicas que todos os métodos de igualdade devem respeitar

O método equals

a assinatura é:

```
public boolean equals(Object o)
```

é importante referir, antes de explicar em detalhe o método, que:

O método equals

a relação de equivalência que o método implementa é:

é **reflexiva**, ou seja `x.equals(x) == true`, para qualquer valor de `x` que não seja nulo

é **simétrica**, para valores não nulos de `x` e `y` se `x.equals(y) == true`, então `y.equals(x) == true`

é **transitiva**, em que para x,y e z, não nulos, se `x.equals(y) == true`, `y.equals(z) == true`, então `x.equals(z) == true`

é **consistente**, dado que para x e y não nulos, sucessivas invocações do método `equals` (`x.equals(y)` ou `y.equals(x)`) dá sempre o mesmo resultado

para valores nulos, a comparação com x, não nulo, dá como resultado false.

quando os objectos envolvidos sejam o mesmo, o resultado é true, ie, `x.equals(y) == true`, se `x == y`

dois objectos são iguais se forem o mesmo, ie, se tiverem o mesmo apontador

caso não se implemente o método equals, temos uma implementação, por omissão, com o seguinte código:

```
public boolean equals(Object object) {  
    return this == object;  
}
```

esqueleto típico de um método equals

```
public boolean equals(Object o) {  
    if (this == o)  
        return true;  
  
    if ((o == null) || (this.getClass() != o.getClass()))  
        return false;  
  
    <CLASSE> m = (<CLASSE>) o;  
    return ( <condições de igualdade> );  
}
```

o método equals da classe Aluno

```
/**  
 * Implementação do método de igualdade entre dois Aluno  
 * Redefinição do método equals de Object.  
 *  
 * @param umAluno aluno que é comparado com o receptor  
 * @return booleano true ou false  
 */  
public boolean equals(Object o) {  
    if (this == o)  
        return true;  
  
    if((o == null) || (this.getClass() != o.getClass()))  
        return false;  
  
    Aluno umAluno = (Aluno) o;  
    return(this.nome.equals(umAluno.getNome()) && this.nota == umAluno.getNota()  
        && this.numero.equals(umAluno.getNumero())  
        && this.curso.equals(umAluno.getCurso()));  
}
```

como é que será o método equals da classe Turma?

quais as consequências de não ter o método equals implementado??

consideremos que Aluno “não tem” equals

o que acontece neste método de Turma?

```
public boolean existeAluno(Aluno umAluno) {  
    boolean resultado = false;  
  
    if (umAluno != null) {  
  
        for(int i=0; i< this.ocupaçao && !resultado; i++)  
            resultado = this.alunos[i].equals(umAluno);  
  
        return resultado;  
    }  
    else  
        return false;  
}
```

O método `toString`

a informação deve ser concisa (*sem acuçar de ecran*), mas ilustrativa

todas as classes devem implementar este método

caso não seja implementado a resposta será:

`getClass().getName() + '@' + Integer.toHexString(hashCode())`

O método `toString`

implementação *normal* de `toString` na classe Aluno

```
/*
 * Implementação do método toString
 * comum na maioria das classes Java
 *
 * @return     uma string com a informação textual do objecto aluno
 */
public String toString() {

    return("Número:" + this.numero + "Nome:" + this.nome + "Nota:" + this.nota);
}
```

o operador “+” é a concatenação de Strings, sempre que o resultado seja uma String

Strings são objectos imutáveis, logo não crescem, o que as torna muito ineficientes

o mesmo método, de forma mais eficiente, na medida em que as concatenações de Strings são muito pesadas

```
/**  
 * Implementação do método toString  
 * comum na maioria das classes Java  
 *  
 * @return      uma string com a informação textual do objecto aluno  
 */  
public String toString() {  
    StringBuilder sb= new StringBuilder();  
  
    sb.append("Número: ");  
    sb.append(this.numero+"\n");  
    sb.append("Nome: ");  
    sb.append(this.nome+"\n");  
    sb.append("Nota: ");  
    sb.append(this.nota+"\n");  
  
    return sb.toString();  
}
```

...completar a classe

Turma

equals

```
/**  
 * Método equals standard do Java.  
 * Utiliza o método privado getAlunos para efectuar a comparação entre  
 * duas instâncias de turma.  
 */  
  
public boolean equals(Object umaTurma) {  
    if (this == umaTurma)  
        return true;  
  
    if((umaTurma == null) || (this.getClass() != umaTurma.getClass()))  
        return false;  
    else {  
        Turma turma = (Turma) umaTurma;  
        return (this.designacao.equals(turma.getDesignacao())  
            && this.capacidade == turma.getCapacidade()  
            && this.ocupacao == turma.getOcupacao()  
            && Arrays.equals(this.alunos,turma.getAlunos()));  
    }  
}
```

nesta versão recorreu-se ao método equals da classe Arrays

é necessário garantir que a remoção de alunos não deixa “lixo” no array alunos

toString

```
/*
 * Método toString por questões de compatibilização com as restantes
 * classes do Java.
 *
 * Como o toString é estrutural e a classe Aluno tem esse método
 * implementado o resultado é o esperado.
 *
 */
public String toString() {
    StringBuffer sb = new StringBuffer();

    sb.append("Designação: "); sb.append(this.designacao+"\n");
    sb.append("Capacidade: "); sb.append(this.capacidade+"\n");
    sb.append("Alunos: "+"\n"); sb.append(this.alunos.toString());

    return sb.toString();
}
```

clone

```
public Turma clone() {
    return new Turma(this);
}
```