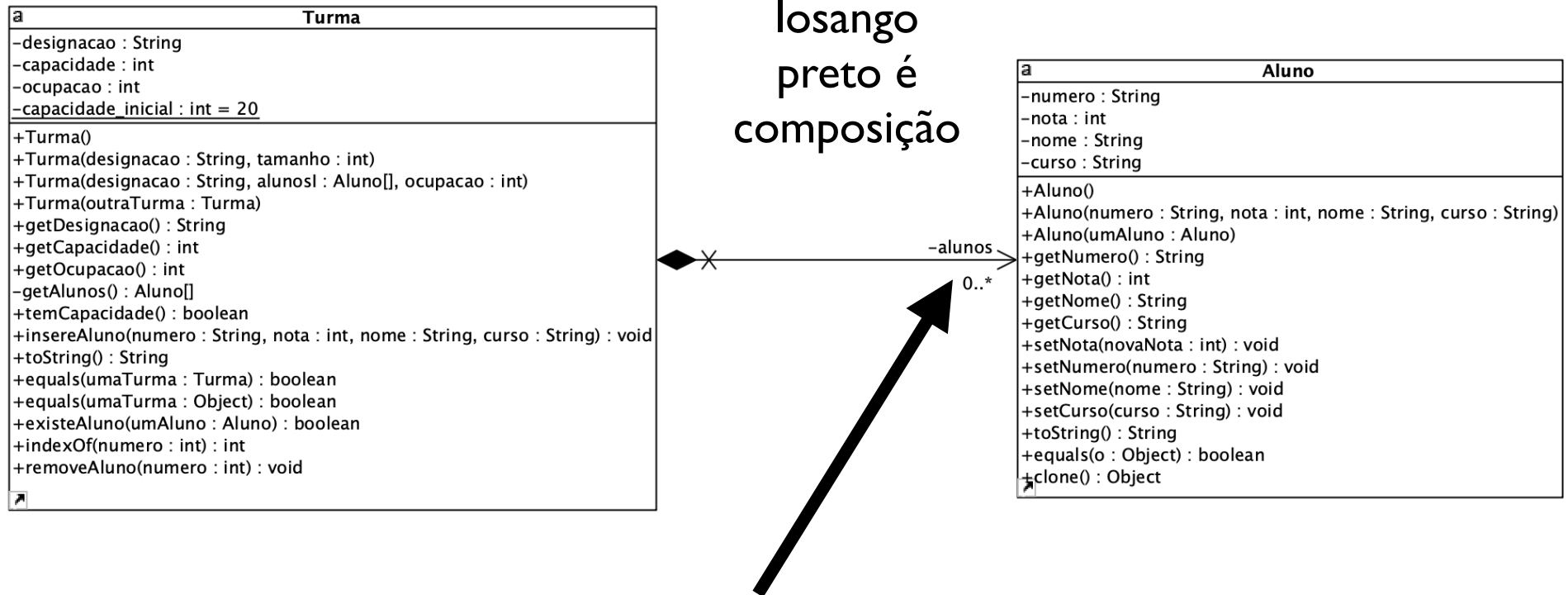


A classe Turma

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

Constructors	
Constructor	Description
<code>Turma()</code>	Constructor por omissão (vazio) para objectos da classe Turma
<code>Turma(java.lang.String designacao, int tamanho)</code>	Constructor parametrizado de Turma.
<code>Turma(java.lang.String designacao, Aluno[] alunosI, int ocupacao)</code>	Constructor parametrizado de Turma em que se envia já os alunos que fazem parte da turma.
<code>Turma(Turma outraTurma)</code>	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 MaterialP00
 * @version 20200216
 * @version 20210304
 */
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.ocupacao;  
}
```

```
/**  
 * 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.
 */

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

- 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.ocupacao] = new Aluno(umAluno);
    this.ocupacao++;
}
```

- Como foi decidido, na fase de concepção, que a arquitectura 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 passá-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

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.ocupacao && !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 entre objectos de uma classe não pode ser codificado a não ser pela classe: abstracção de dados
 - existe 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;  
}
```

- template 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.ocupacao && !resultado; i++)  
            resultado = this.alunos[i].equals(umAluno);  
  
        return resultado;  
    }  
    else  
        return false;  
}
```